Proyecto Fin de Carrera

Desarrollo de un servicio REST e interfaz gráfica para un sistema de gestión de contenido dinámico

Autor: Javier Rojano Broz
Tutor: Francisco J. Santana Pérez
Mayo de 2017

Agradecimientos

El desarrollo de este Proyecto de Fin de Carrera ha supuesto para mi un reto mayor de lo esperado y si no he fracasado ha sido gracias al apoyo de diversas personas, en especial mi familia, que siempre estuvo a mi lado y me motivó para continuar.

Quiero hacer una mención especial a mi mujer Ana y a mis hijos Alvaro y Eva. Durante la elaboración de este proyecto han pasado cosas maravillosas en mi vida personal, como el nacimiento de mi primer hijo y el embarazo del segundo. A pesar de ello, y de todo el trabajo que la maternidad conlleva, mi mujer ha sido la principal fuente de apoyo durante los momentos mas duros. Sin una persona como ella a mi lado este proyecto nunca se hubiera podido llevar acabo.

Por último deseo expresar mi mas sincero agradecimiento al profesor Francisco J. Santana Pérez, tutor de este proyecto, por la ayuda y dedicación mostrada durante el desarrollo del mismo.

1. Introducción

En este Proyecto de Fin de Carrera se abordará el desarrollo de un **Sistema de Gestión de Contenido dinámico a través de un servicio web REST y de una interfaz gráfica de administración.**

Para su desarrollo se empleado la pila o stack de aplicaciones conocida con el acrónimo MEAN (MongoDB - Express - AngularJS - Node.JS).

1.1 Definiciones

Sistemas de Gestión de Contenidos

Un sistemas de gestión de contenidos o CMS, por sus siglas en inglés (Content Management System) es una aplicación informática que facilita la administración de contenidos. Un CMS ofrece una interfaz intuitiva para que usuarios sin conocimientos de programación puedan llevar a cabo las tareas de administración de contenidos incluyendo las tareas de publicación, edición y borrado de los mismos.

Generalmente los CMS trabajan contra una base de datos, de modo que el editor simplemente actualiza una base de datos, incluyendo nueva información o editando la existente.

Servicio Web

Existen múltiples definiciones sobre lo que son los Servicios Web (en inglés, Web Service o Web services), lo que muestra su complejidad a la hora de dar una adecuada definición que englobe todo lo que son e implican. Una posible definición sería hablar de ellos como un conjunto de aplicaciones o de tecnologías con capacidad para interoperar en la Web. Estas aplicaciones o tecnologías intercambian datos entre sí con el objetivo de ofrecer unos servicios. Los proveedores ofrecen sus servicios como procedimientos remotos y los usuarios solicitan un servicio llamando a estos procedimientos a través de la Web.

Estos servicios proporcionan mecanismos de comunicación estándares entre diferentes aplicaciones, que interactúan entre sí para presentar información dinámica al usuario.

Para proporcionar interoperabilidad y extensibilidad entre estas aplicaciones, y que al mismo tiempo sea posible su combinación para realizar operaciones complejas, es necesaria una arquitectura de referencia estándar.

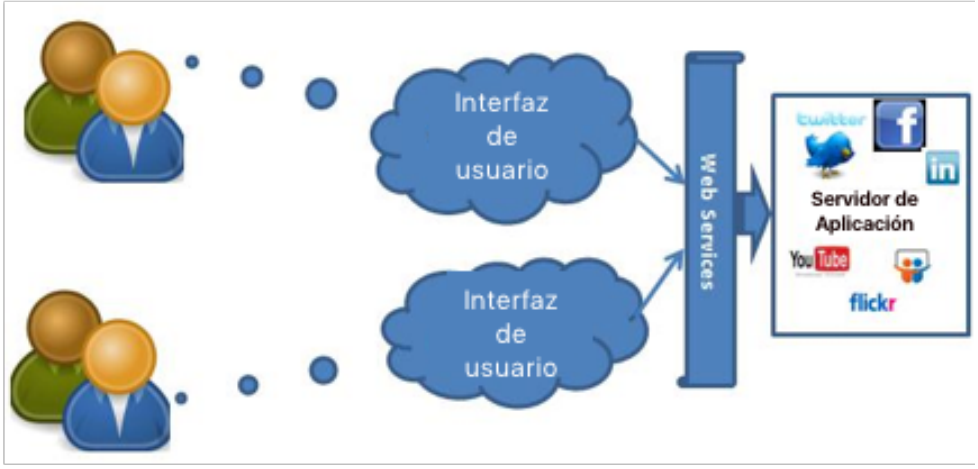


Gráfico 1: Web Service

Las principales ventajas de los servicios web son:

- Aportan interoperabilidad entre aplicaciones de software independientemente de sus propiedades o de las plataformas sobre las que se instalen.

- Los servicios Web fomentan los estándares y protocolos basados en texto, que hacen más fácil acceder a su contenido y entender su funcionamiento.

- Permiten que servicios y software de diferentes compañías ubicadas en diferentes lugares geográficos puedan ser combinados fácilmente para proveer servicios integrados.

API (Application Programming Interface)

La interfaz de programación de aplicaciones, abreviada como API (del inglés: Application Programming Interface), es el conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

REST (representational state transfer)

La Transferencia de Estado Representacional (REpresentation State Transfer - REST) describe un estilo arquitectónico de sistemas en red como, por ejemplo, aplicaciones Web. El término fue utilizado por primera vez en el año 2000 durante una disertación doctoral por Roy Fielding, uno de los principales autores de la especificación HTTP. REST está comprendida por una serie de limitaciones y principios arquitectónicos. Si una aplicación o diseño cumple con esas limitaciones y principios, se considera RESTful.

Uno de los principios REST de mayor importancia para las aplicaciones Web es que la interacción entre el cliente y el servidor no tiene estado entre solicitudes. Cada solicitud del cliente al servidor debe contener toda la información necesaria para comprender la solicitud. El cliente no se dará cuenta si el servidor debe reiniciarse en ningún momento entre las solicitudes. Asimismo, las solicitudes sin estado pueden ser respondidas por cualquier servidor disponible, lo cual resulta apropiado en un entorno como la computación en nube.

En el extremo del servidor, el estado y la funcionalidad de la aplicación se dividen en recursos. Un recurso es un elemento de interés, una identidad conceptual que se expone a los clientes. Algunos ejemplos de recursos son: objetos de aplicaciones, registros de bases de datos, algoritmos, etc. Cada recurso es de acceso único a través de una URI (Universal Resource Identifier – identificador de recursos universal). Todos los recursos comparten una interfaz uniforme para la transferencia de estados entre cliente y servidor. Se usan métodos estándar HTTP como GET, PUT, POST y DELETE. El motor del estado de la aplicación es Hypermedia y las representaciones de recursos se interconectan mediante hipervínculos.

Otro principio REST importante es el de sistema por capas, el cual implica que un componente no puede ver más allá de la capa inmediata con la cual interactúa. Al restringir el conocimiento del sistema a una sola capa, se impone un límite en la complejidad del sistema en general, promoviendo así la independencia de los sustratos.

Las limitaciones arquitectónicas REST, aplicadas como un todo, genera una aplicación que logra escalar sin problemas a grandes cantidades de clientes. También se reduce la latencia en la interacción entre clientes y servidores. La interfaz uniforme simplifica la arquitectura

general del sistema y mejora la visibilidad de las interacciones entre subsistemas. REST simplifica la implementación tanto para el cliente como para el servidor.

En la siguiente figura podemos observar un ejemplo de diseño de una arquitectura REST:

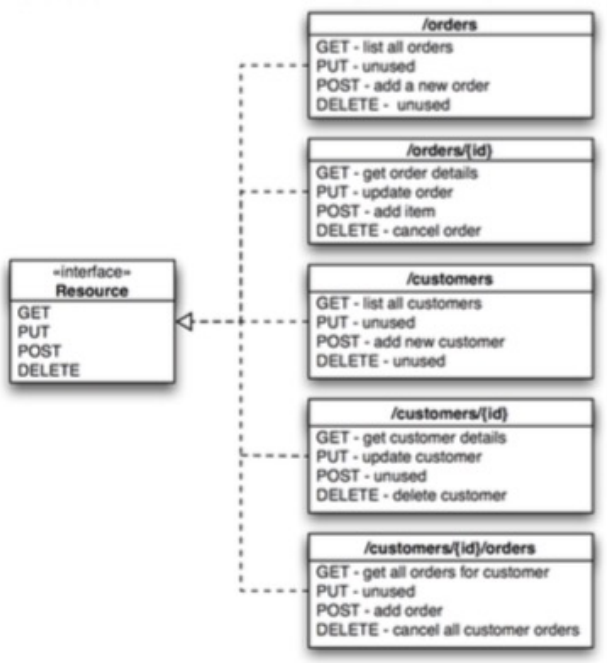


Gráfico 2: arquitectura REST

En una API RESTful intervienen tres componentes: la aplicación, la API y el cliente. El siguiente gráfico ilustra cómo interactúan estos tres componentes.

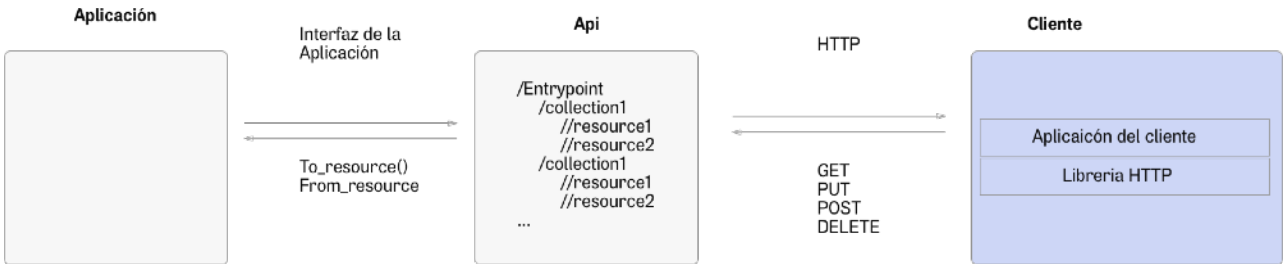


Gráfico 3: Componentes API RESTfull

2. Estado del Arte y Objetivos

2.1 Historia

Los primeros CMS surgieron para administrar documentos y ficheros locales pero en la actualidad están principalmente diseñados para distribuir el contenido a través de la web.

A principios de la década de los 90, coincidiendo con el inicio de la web los sistemas de gestión de contenidos eran herramientas desarrolladas a medida por grandes corporaciones que disponían de sitios web que debían ser actualizadas diariamente o varias veces por día, y donde además, las personas que editaban la información no tenían conocimientos avanzados de informática.

En 1995 Portland Pattern Repository publicó Wiki Wiki Web (en hawaiano rápido y sencillo) considerado como el primer CMS gratuito o de código abierto. El software y la web fueron desarrollados por Ward Cunningham tomando como base de partida hyperCard, creado por Apple Systems a mediados de los 80 y considerado por muchos como el primer CMS.

Durante la segunda mitad de la década de los 90 surgieron gran cantidad de sitios webs wikis basados en perl y destinados a áreas de conocimiento muy específicas. También surgieron los primeros sistemas de gestión de contenidos gratuitos basados en PHP como Php Nuke, PostNuke y PhpWiki. A pesar de que supusieron un gran avance sólo fueron adoptados por personas con un perfil muy técnico debido a su complejidad tanto en la instalación como en la administración.

En el año 2000 llegó primera gran revolución en lo que a tecnología de sistemas CMS se refiere. La publicación de php4, un sistema generador de html gratuito y de fácil aprendizaje, junto con la disponibilidad de un sistema operativo estable (Linux), un servidor web gratuito (Apache) y una sistema de gestión de base de datos gratuito (MySql) dotaron a los desarrolladores web de las herramientas necesarias para desarrollar sitios webs complejos con un coste reducido y sin necesidad de años de aprendizaje.

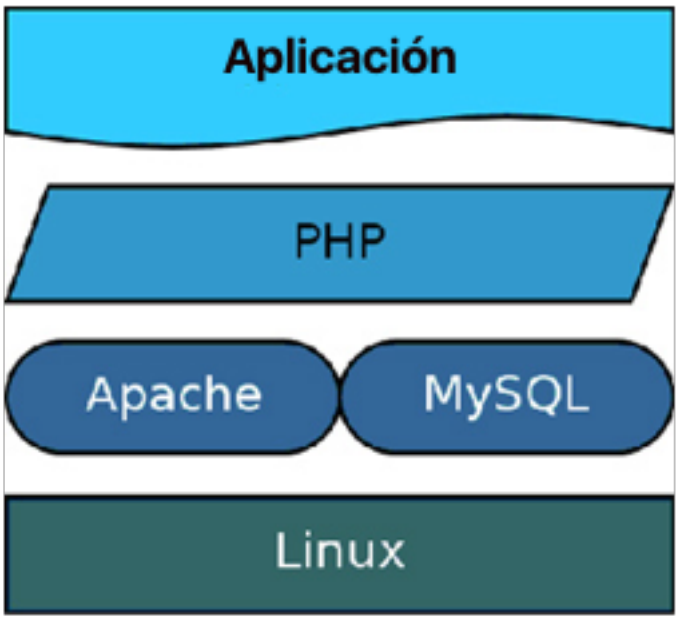


Gráfico 4: pila de aplicaciones LAMP

El conjunto de estas 4 herramientas pasó a denominarse **LAMP (Linux Apache MySQL PHP)** y fue adoptado por la mayoría de los CMS Open Source más populares Mambo/ Joomla, Wordpress, Drupal etc.

La popularidad de LAMP se ha mantenido durante años pero en la actualidad están surgiendo nuevas tecnologías como las bases de datos no Sql y NodeJS que están haciendo tambalear su hegemonía en el desarrollo web. De entre todas estas nuevas tecnologías quizás la que está gozando de mayor auge es la pila o stack de aplicaciones conocida con el acrónimo **MEAN (MongoDB - Express - AngularJS - Node.JS)** que se caracteriza por el desarrollo end-to-end usando JavaScript tanto en el frontend, backend y la base de datos.

Sin entrar en definir completamente cada una de las tecnologías que lo componen nos encontramos con:

- **MongoDB** u otra base de datos no relacional como base de datos que almacena documentos JSON.
- **Express** como web framework basado en Node.js que nos permite crear API REST, por ejemplo
- **AngularJS** como framework para crear la parte cliente de la aplicación en formato Single Page.

- **Node.js** como framework JavaScript basado en V8 que proporciona funcionalidad para nuestra aplicación bajo un modelo asíncrono de eventos.

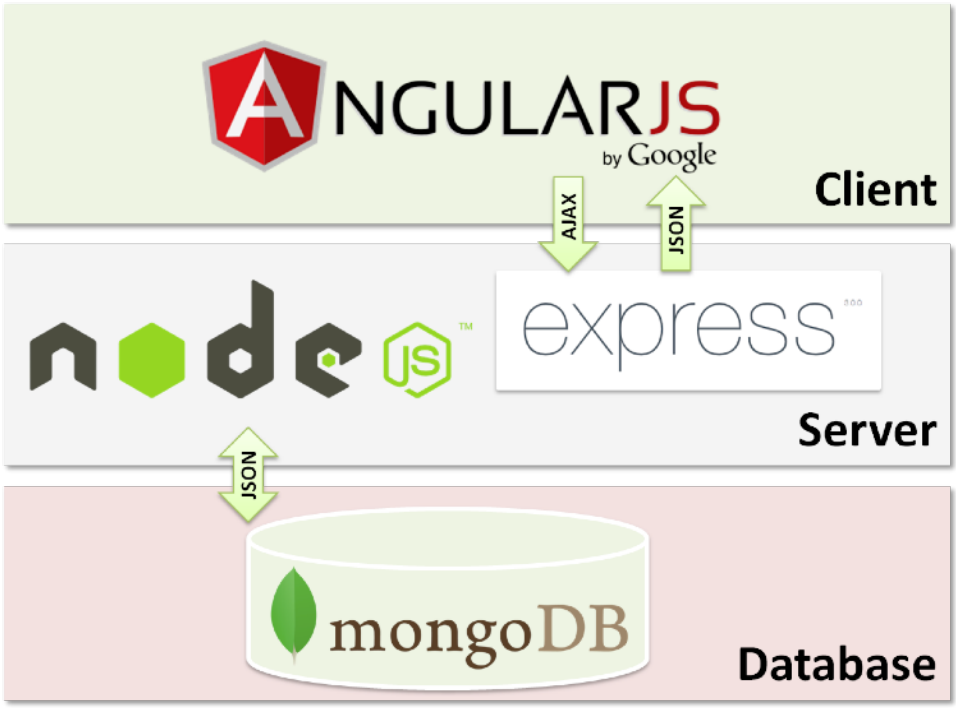


Gráfico 5: Pila de aplicaciones MEAN

2.2 Motivación

Como desarrollador de aplicaciones web me he encontrado a lo largo de mi carrera profesional un gran número de proyectos pequeños y medianos que compartían un mismo conjunto de requisitos: página web autogestionable, en la que mostrar contenido multimedia estructurada en diferentes secciones o categorías. El cliente para este tipo de proyectos suele ser una empresa de marketing o publicidad por lo que el diseño ya estaba definido y aprobado antes de comenzar el desarrollo.

Inicialmente abordé este tipo de encargos con desarrollos a medida para cada proyecto pero pronto comencé a usar Sistemas de Gestión de Contenidos de código abierto basados en el modelo LAMP (Joomla/mambo, wordpress, etc).

Tras estudiar las opciones disponibles en el mercado elegí Mambo (Que posteriormente pasó a denominarse Joomla) como plataforma para este tipo de desarrollos pero, pronto detecté que el coste de mantenimiento era demasiado elevado debido a los continuos problemas de seguridad y por la dificultad de actualización a nuevas versiones.

Otros problemas derivados del uso de Sistemas de Gestión de Contenidos:

- **Gestión de múltiples servidores.** En la mayoría de los casos el cliente cuenta con un servidor en el que quiere instalar la nueva app esto obliga a trabajar con servidores con distintos Sistemas Operativos y diferentes tecnologías. también implica un mayor costo a la hora de actualizar a nuevas versiones.

- **Limitaciones en el servidor del cliente.** El servidor del cliente debe cumplir una serie de requisitos mínimos para poder instalar la última versión del Sistema de Gestión de Contenido en cuestión. No todos los clientes tienen disponible en sus servidores los requerimientos mínimos para instalar un determinado CMS de código abierto.

- **Elevado consumo de recursos.** Los Sistemas de Gestión de Contenidos disponibles en el mercado han evolucionado considerablemente hasta convertirse en herramientas muy potentes y personalizables. Todo esta potencia implica un incremento en el consumo de recursos del servidor y repercute negativamente en el tiempo de carga de la página. El perfil de cliente para el que se ha orientado este proyecto busca una página web sencilla en la que mostrar información corporativa, por tanto no está justificado el uso de herramientas que sobrecarguen el sistema.

- **Uso de plugins desarrollados por terceros.** Una de las características más destacables de los Sistemas de Gestión de Contenido más populares es la facilidad para ampliar su funcionalidad mediante la instalación de plugins. Prácticamente se puede encontrar un plugin para cualquier cosa que se nos ocurra, por tanto es muy probable que el administrador del sitio se sienta tentado y acabe instalando gran cantidad de plugins y es cuestión de tiempo que alguno de ellos afecte al funcionamiento normal del sistema o que incluso acabe bloqueando por completo. Con cada instalación de plugin se incremente el riesgo de fallo y si no se actualizan con frecuencia el riesgo es aún mayor.

- **Elevado coste de implementar desarrollos a medida.** En ocasiones los clientes tienen necesidades específicas que no pueden implementarse con la funcionalidad básica del Sistema de Gestión de Contenido ni con los plugins disponibles. Desarrollar funcionalidad a medida requiere de un conocimiento previo sobre el o los lenguajes de programación, bases de datos y conocer la referencia de funciones que el sistema ofrece para ampliar su funcionalidad.

- **Panel de control complicado para usuarios inexpertos**. Como se ha comentado con anterioridad los CMS son cada vez más potentes y mas personalizables. Esto, que a priori parece una ventaja, se convierte en una barrera de entrada para los usuarios menos expertos, ya que no disponen de los conocimientos necesarios para enfrentarse a una herramienta tan complicada. Para este tipo de usuarios los paneles de control de estas herramientas pueden resultar intimidantes.

- **Dificultad para sitios con múltiples idioma**. La mayoría de los Sistema de Gestión de contenidos no dan soporte para varios idiomas en su funcionalidad básica teniendo que hacer uso de plugins de terceros. En la práctica la forma en la que estos plugins resuelven el problema complica aún más el proceso de creación de contenido.

- **Dificultad de adaptación del diseño aportado por el cliente**. Otra de las características más destacadas de los Sistemas de Gestión de Contenidos más populares es la posibilidad de personalización mediante plantillas. Existen miles de plantillas disponibles, ya sean comerciales o gratuitas. El problema viene cuando el cliente impone un diseño que no está basado en una plantilla. El coste de creación de una plantilla desde cero suele ser más costoso que un desarrollo a medida de la aplicación completa.

Recientemente he experimentado estos problemas en un servidor que administro. Se trata de un servidor cloud linux con la distribución CentOS 5 con Plesk 9 (64 bit) con las siguientes características de hardware:

- Procesador virtual de dos núcleos
- 2gb de Ram
- 300gb de disco duro

El principal uso de este servidor es como servidor web (Apache2 + php 5.6) y servidor de base de datos (Mysql 5.6) y durante más de 4 años ha estado funcionando sin problemas para servir simultáneamente más de 20 sitios webs basados con programación a medida basada en LAMP y 3 aplicaciones MEAN. Los problemas comenzaron tras la instalación de dos wordpress con bastante tráfico de visitas diaria.

El primer wordpress se instaló en enero de 2016 y ha estado recibiendo una media de 5365 peticiones http, con una media de 200 usuarios únicos y con picos de 40 usuarios simultáneos:

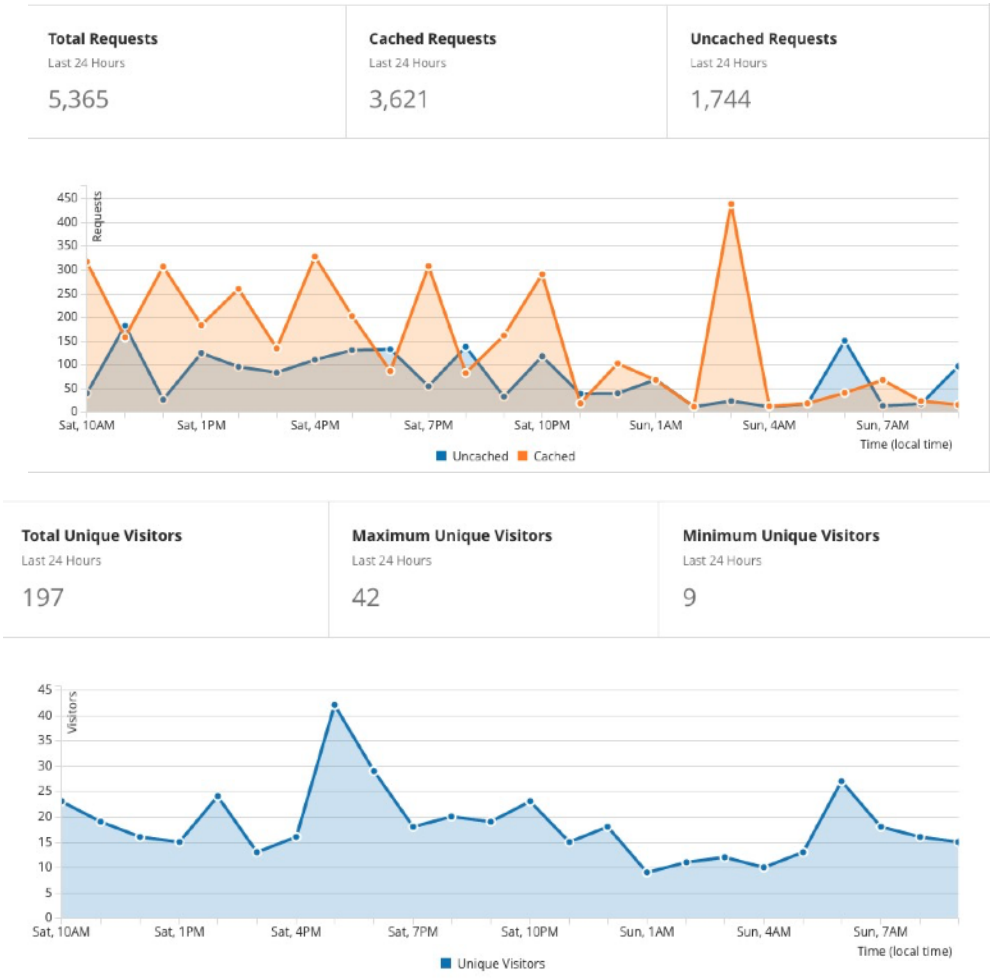


Gráfico 6: estadísticas wordpress 1

El segundo wordpress se instaló en mayo de 2016 y recibió una media de 2300 peticiones diarias con 215 usuarios únicos con picos de 25 usuarios simultáneos.

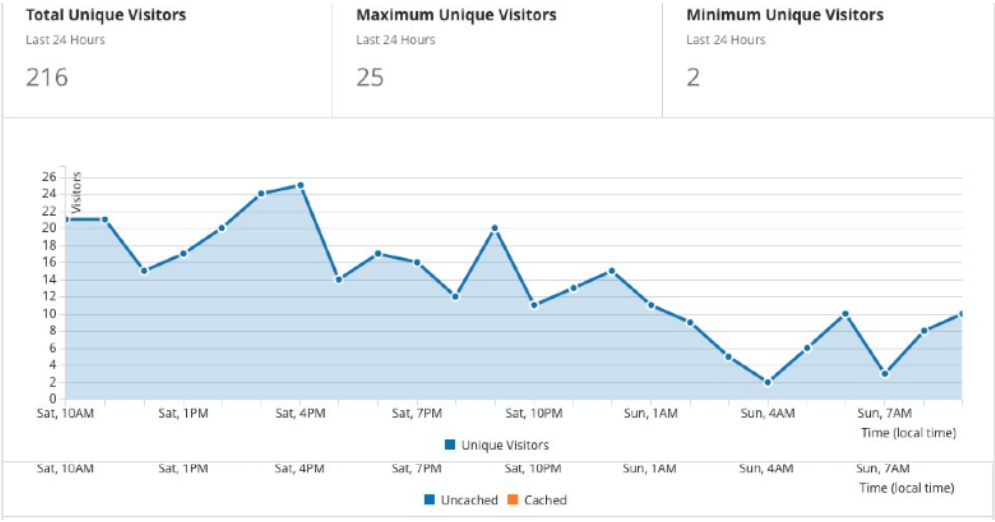


Gráfico 7: estadísticas Wordpress 2

Tras la instalación del primer Wordpress el sistema comenzó a comportarse de manera inestable provocando que en determinados momentos el servidor Apache respondiera con un error 500. Tras revisar los logs detecté que el problema se debía a que apache no tenía memoria suficiente:

```
Cannot allocate memory: couldn't create child process
```

No obstante los errores eran poco frecuentes y el sistema era capaz de recuperarse solo en cuestión de pocos minutos.

Con la instalación del segundo Wordpress los problemas se agravaron notablemente provocando que el servidor estuviera casi el 90% del tiempo sin dar servicio.

Gráfico 8: Uptime servidor

En el gráfico anterior se puede apreciar cómo se deterioraron las estadísticas de “uptime” del servidor

En junio de 2016 decidí migrar el primer sitio Wordpress a un nuevo servidor, lo que mejoró el rendimiento del servidor.

Para evitar que estos problemas de memoria se volvieran a repetir en el nuevo servidor se sustituye la pila de aplicaciones LAMP por la pila de tecnología denominada LEMP (Nginx, MariaDB (MySQL), PHP).

Aunque en la actualidad están surgiendo algunos CMS basados en node y mongo JS ninguno de ellos se basa en el uso de web services. Algunos ejemplos de estos CMS son:

Nombre	Link
KeystoneJS	http://keystonejs.com/
Apostrophe	http://apostrophenow.org/

La experiencia previa con sistemas de gestión de contenidos basados en LAMP y la dificultad para encontrar un cms de código abierto basado en la pila de aplicaciones MEAN que haga uso de webservices han sido las principales motivaciones para el desarrollo de este proyecto.

2.3 Objetivos

El principal objetivo de este proyecto es la construcción de un sistema que permita la gestión de contenido y que resuelva los problemas planteados por los Sistemas de Gestión de Contenido de código basados en LAMP.

Para el desarrollo del nuevo sistema será necesario la adquisición de conocimientos de las nuevas tecnologías que se van a emplear.

El sistema resultante debe reunir las siguientes características:

- Uso de arquitectura cliente/servidor.
- Uso de la pila de aplicaciones MEAN
- Facilidad para escalar
- Implementación de una interfaz API REST que permita que diferentes tipos de clientes accedan a los datos para su consumo como para manipulación.
- La interfaz gráfica de administración debe ser lo suficientemente sencilla como para que usuarios sin conocimientos avanzados de informática puedan hacer uso de ella.

Una vez desarrollado y probado el sistema se procederá a su puesta en marcha en un servidor de producción para su uso con datos reales mediante la creación de varios sitios web.

3. Recursos necesarios

Para la implementación del proyecto se ha optado por un modelo basado en dos entornos diferenciados, el entorno de desarrollo y pruebas y el entorno de producción.

Un entorno es un espacio técnico que posee un alcance bien definido y respetado. Una buena práctica, común en equipos de desarrollo ágil, es asegurar que los desarrolladores poseen su propio entorno donde trabajar. La principal ventaja de los entornos es que ayudan a reducir los riesgos debido a errores técnicos que puedan afectar de forma adversa a un grupo de personas mayor al absolutamente necesario.

El modelo de desarrollo, pruebas y producción es esencial para proveer los controles y el balance necesario para ejecutar un entorno de producción de alta disponibilidad de forma eficiente.

La sincronización entre los diferentes entornos se realizará mediante el uso de un sistema de control de versiones.

3.1 Entorno de desarrollo y pruebas:

Es el entorno de trabajo para desarrolladores individuales o pequeños equipos de desarrolladores. Trabajando de forma aislada con el resto de las capas, los desarrolladores pueden probar cambios radicales en el código sin afectar de forma adversa al resto del equipo de desarrollo. Estos entornos suelen estar ubicados directamente en las estaciones de trabajo de cada desarrollador.

En nuestro caso e entorno de desarrollo se ejecutará en un ordenador portátil con las siguientes características:

- MacBook Pro (13 pulgadas, finales de 2011)
- Sistema Operativo: OSX El Capitán
- Procesador: 2.4 GHz Intel Core i5
- Memoria: 16Gb 133Mhz DDR3

Para poder emular el servidor REST en nuestro ordenador debemos tener instalado:

- **MongoDB Community Edition**. Para la instalación de mongoDb se puede optar por el uso del gestor de paquetes Homebrew o por la instalación manual.
- **Node.** Para instalar node hemos optado por usar Node Version Manager, pero quien lo prefiera puede descargar el código fuente o usar el instalador correspondiente a su sistema operativo. Node Version Manager es un script Bash que nos permite instalar y gestionar diferentes versiones de node.

La interfaz gráfica o panel de control está implementada haciendo uso de Angular JS, para instalar las dependencias de esta aplicación haremos uso del gestor de paquetes de node npm (Node Package Manager).

- **Yeoman**. El propósito principal de esta herramienta es la agilización del proceso del inicio del desarrollo, mediante la construcción de un esqueleto bastante completo para el tipo de aplicación web que estés haciendo, este procedimiento también es conocido como scaffolding.

Yeoman se apoya de tres herramientas para desarrollar su cometido: La herramienta para generar el scafolding (yo), automatizador de tareas (Gulp, Grunt, etc) y el gestor de paquetes (npm, bower).

- **Bower**. Bower es un gestor de dependencias para el desarrollo web frontend que facilita la tarea de instalar y mantener al día librerías en nuestros proyectos. Es un programa basado en node.js que se ejecuta desde consola y que tiene un sencillo API de comandos útiles para realizar tareas de mantenimiento y administración de paquetes necesarios para construir un proyecto web, concretamente la parte del lado del cliente.

- **Grunt.** Grunt es una librería JavaScript que nos permite configurar tareas automáticas como minificación, compilación, validación de sintaxis, pruebas unitarias, observar cambios de tus archivos, concatenación de archivos, copiado de archivos de una ruta a otra, borrado de archivos, generar documentación, crear sprites, etc.

Para poder depurar el código y realizar pruebas hemos activado un servidor web local basado en node.js

El control de versiones tanto del servidor REST como de la interfaz gráfica se realizará haciendo uso de git. Esto nos permitirá automatizar el paso de desarrollo a producción.

Control de versiones: Git, bitbucket y SourceTree

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación.

A pesar de que este proyecto no va a estar desarrollado por un equipo sino por una sólo persona he considerado oportuno usar Git como sistema de gestión de versiones para poder tener constancia de los cambios que se van realizando en cada uno de los archivos y acceso fácil a un histórico de copias de seguridad.

Git

Git es el sistema de control de versiones más utilizado en la actualidad. Git es un proyecto de software libre originalmente desarrollado por Linus Trovalds en 2005.

La principal ventaja de GIT es que hace uso de una arquitectura distribuida de manera que la copia local de cada desarrollador actúa como repositorio conteniendo el historial completo de cambios. Permite la fusión del código.

Aparte de ser distribuido, Git ha sido desarrollado teniendo en cuenta la eficiencia, seguridad y la flexibilidad.

Bitbucket

Bitbucket es un servicio online para almacenar repositorios, compatible con Git y con Mercurial. La razón principal por la que he elegido Bitbucket frente a otras opciones,

como por ejemplo Github es porque la versión gratuita permite almacenar repositorios privados.

SourceTree

SoourceTree es una aplicación de escritorio para manejar repositorios Git y Mercurial desde una interfaz gráfica. Permite la gestión de repositorios locales y remotos.

Otras Herramientas.

Sublime Text 2. Sublime Text, es un sencillo pero potente editor de código disponible para Linux, Windows y Mac OS. Aunque se trata de un software comercial, se puede evaluar sin ningún tipo de restricción temporal ni funcional. Es un editor de código muy ligero, rápido , estable y fácilmente ampliable gracias a su gestor de paquetes y al gran número de plugins disponible.

Studio 3T. Se trata una herramienta multiplataforma con la que podemos administrar gráficamente nuestras bases de datos mongo.

Sin duda, el shell mongo es muy potente, y con él nos basta y nos sobra para iniciarnos en su uso y trabajar normalmente con las colecciones de nuestras bases de datos. Sin embargo, cuando ya hemos pasado cierto tiempo utilizándolo nos damos cuenta de que tiene algunas deficiencias. Por ejemplo, necesitamos abrir un terminal por cada sesión iniciada en Mongo, perdemos los resultados de nuestra consultas sin posibilidad de hacer scroll, etc.

Studio 3T integra en un entorno gráfico con toda la funcionalidad del shell Mongo y añadiendo otras nuevas como:

- Múltiples conexiones a las bases de datos, separadas por pestañas.
- Resaltado de sintaxis y autocompletado de código.
- Distintos modos de visualización.

Postman. Es una herramienta que funciona como una extensión de Google Chrome y nos permite construir y gestionar de una forma cómoda nuestras peticiones a servicios REST (Post, Get, etc). Su manejo es realmente intuitivo ya que simplemente tenemos que definir la petición que queremos realizar.

Además nos permite crear colecciones para guardar nuestras peticiones REST categorizadas.

3.2 Entorno de Producción:

Es el entorno donde trabajan los usuarios finales y se trabaja con los datos de negocio. En nuestro caso el entorno de producción está compuesto por un servidor cloud con las siguientes características:

- Sistema operativo:Linux
- Distribución: Debian
- Versión: Debian 8
- Arquitectura:64 bits
- CPU:4 vCore
- RAM:8 GB
- SSD:160 GB

Aunque el sistema está preparado para poder instalar la interfaz gráfica y el servidor REST en servidores diferentes, para lo propósitos de este proyecto se ha optado por usar el mismo servidor con el objetivo de ahorrar costes.

Para poder ejecutar el servidor Rest debemos instalar en el servidor:

- Git
- node
- mongoBd

Para poder ejecutar la interfaz web debemos tener disponible en el servidor de producción un servidor web. Tras estudiar las principales alternativas disponibles en el mercado se ha optado por instalar NGINX.

¿Por qué NGINX?

Apache y Nginx son dos de los servidores webs de código abierto más usados en la actualidad. Juntos son responsables de servir el 50% del tráfico que circula por internet.

Antes de entrar a enumerar las diferencias entre Apache y Ngnix, vamos a presentar una pequeña introducción a cada uno de ellos:

Apache. El servidor HTTP apache fue creado por Robert McCool en 1995 y desarrollado bajo la dirección de Apache Software Foundation desde 1999. Ha sido durante décadas el servidor web más popular y por tanto una de sus principales ventajas es que es fácil encontrar documentación y soporte.

Los administradores de sistemas que optan por Apache lo hacen por su flexibilidad, potencia y su amplia popularidad. Además se puede extender mediante un sistema de módulos dinámico.

Nginx. En 2002, Igor Sysoev comenzó a trabajar en Nginx como respuesta al problema conocido como C10K, que consistía en el desafío que suponía para los servidores atender a 10 mil conexiones simultáneas como consecuencia de las nuevas aplicaciones web. La primera versión pública vio la luz en 2004.

La popularidad de Nginx ha crecido exponencialmente desde entonces gracias a su bajo consumo de recursos y a su escalabilidad con unos requisitos de hardware mínimos. Ngnix es muy potente a la hora de servir contenido estático y está diseñado para delegar las peticiones dinámicas a otras piezas de software mejor diseñadas para este objetivo.

La principal diferencia entre ambos sistemas es la manera en que estos manejan las conexiones y el tráfico. En Apache para cada cliente hay un hilo que es totalmente independiente y se dedica a servir a ese hilo. Esto puede hacer que existan problemas de bloqueo cuando el proceso está a la espera para ser completado para liberar los recursos

(memoria, CPU) en el disco duro. Además, la creación de procesos separados consume más recursos.

En Nginx, la solución para resolver el problema anterior es el uso por eventos, asíncrono, sin bloqueo y la arquitectura de un solo subproceso. A diferencia de Apache Ngnix no que crea un nuevo proceso para cada solicitud.

Varios resultados de evaluación comparativa indica que, en comparación con Apache, Nginx es extremadamente rápido para servir páginas estáticas.

Debido a la arquitectura de nuestros proyecto en el que el contenido dinámico es servido mediante peticiones a un servidor REST, Ngnix es la opción idónea como servidor web.

PM2

PM2 es un gestor de procesos de producción para las aplicaciones Node.js que tiene un balanceador de carga incorporado. PM2 permite mantener siempre activas las aplicaciones y volver a cargarlas sin ningún tiempo de inactividad, a la vez que facilita tareas comunes de administrador del sistema. PM2 también permite gestionar el registro de aplicaciones, la supervisión y la agrupación en clúster.

Política de cortafuegos.

Un firewall o cortafuegos es un sistema que define, por medio de una política de reglas, qué tipo de tráfico se permite en una red y en el que podemos indicar el tipo de conexión autorizada, definiendo los protocolos, puertos y las direcciones IP permitidas.

Un firewall es, por tanto, una de las mejores medidas que podemos implementar para securizar nuestro proyecto. Para este proyecto hemos optado por una política restrictiva cerrando todo el tráfico desde y hacia nuestro servidor , abriendo el tráfico única y exclusivamente para aquellos servicios que se vayan utilizar. En nuestro caso:

Protocolo	Desde	Hasta	Descripción
TCP	80	80	**HTTP**
TCP	443	443	**HTTPS**

Protocolo	Desde	Hasta	Descripción
TCP	22	22	**SSH**
TCP	21	21	**FTP**
TCP	27017	27017	**MongoDB**

Implantación de un CDN mediante Cloudflare

Como capa adicional para mejorar el rendimiento y la seguridad del servidor se configurado todo el sistema para que haga uso de cloudflare.

Cloudflare (EN) es un servicio de CDN en proxy inverso. CloudFlare asegura y acelera tu página al actuar como un proxy entre tus visitas y el servidor. Con CloudFlare proteges tu sitio web contra visitas maliciosas, ahorras ancho de banda y reduces el tiempo promedio de carga.

Las principales características de cloudflare son:

- CDN Distribuye tu contenido alrededor del mundo para estar más cerca de tus visitas.
- Optimización Los contenidos son optimizados, comprimiendo CSS, JavaScripts, y similares, para que se sirvan más rápidos..
- Seguridad Protege tu web de muchas de las amenazas en la red: spammers, inyección SQL, ataques DDoS.
- Analítica Más información sobre la realidad a la que se enfrenta tu web: amenazas, crawlers, buscadores...

4. Análisis, Diseño e Implementación

4.1 Requisitos funcionales y no funcionales

En este apartado veremos los diferentes requisitos que se han encontrado para el desarrollo del proyecto.

4.1.1 Requisitos funcionales

A continuación se describe los requisitos funcionales que debe cumplir el sistema.

Referencia	Descripción
RF001	Los usuarios accederán al sistema haciendo uso de sus credenciales: email y contraseña.
RF002	Los usuarios podrán gestionar las categorías. La gestión de categorías engloba las tareas de creación, edición y eliminación.
RF003	Los usuarios podrán gestionar las categorías. La gestión de categorías engloba las tareas de creación, edición y eliminación.
RF004	La pantalla principal deberá mostrar un resumen del contenido almacenado en el sistema.
RF005	Todas las categorías deberán tener un nombre y una descripción así como una serie de valores de configuración para indicar si se permite que los usuarios añaden nuevo contenido o si se permite a los usuarios borrar contenidos de la categoría.
RF006	A parte de los campos obligatorios (RF005) los usuarios podrán definir los elementos de contenido que tendrá la categoría. Los elementos de contenidos están predefinidos en el sistema.
RF007	Una categoría podrá tener tantos elementos de contenido como el usuario quiera.
RF008	El sistema deberá ser multiidioma. El usuario deberá poder añadir contenidos en varios idiomas.
RF009	El sistema contará con dos roles de usuarios. Usuarios administradores y usuarios redactores. Los usuarios administradores podrán administrar categorías, contenidos y usuarios. Los usuarios redactores sólo podrán editar contenido.

4.1.2 Requisitos no funcionales

Los requisitos no funcionales del sistema son aquellos que describen cualidades o criterios que sirven para juzgar el funcionamiento general del sistema, en lugar de las funcionalidad específicas. Estos requisitos comprenden las características de seguridad, disponibilidad, accesibilidad, escalabilidad, extensibilidad, tolerancia a fallos, usabilidad, etc

Referencia	Descripción
RFN001	El servidor debe discriminar a los usuarios identificados en el sistema limitando el acceso a los recursos en función de su rol.
RFN002	La interfaz de usuario debe ser accesible desde los principales navegadores web.
RFN003	Las contraseñas se deben almacenar haciendo uso de algoritmo de "un sólo sentido" de forma que no se puedan descifrar.
RFN004	El modelo de datos debe permitir que las categorías tengan una estructura flexible. Es decir una categoría podrá contener cualquier combinación componentes (field types)
RFN005	El desarrollo del sistema deberá seguir un diseño modular que permita su mantenimiento y extensión
RFN006	La autenticación se realizará mediante un sistema basado en tokens.

4.2 Diseño de la arquitectura del sistema

En este apartado se establecerá la arquitectura del sistema a alto nivel identificando los elementos hardware y software.

El diseño arquitectónico escogido se basa en una arquitectura cliente-servidor donde los diferentes clientes se comunicarán con el servidor haciendo uso de una api REST.

Existirán dos tipos de clientes: La interfaz gráfica de administración o Backend y el cliente público o frontend.

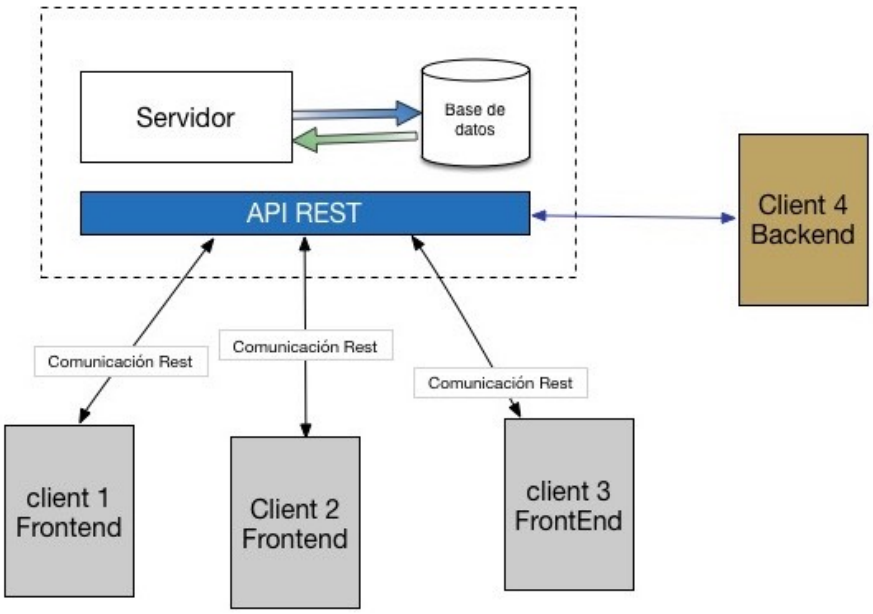


Gráfico 9: Diseño Arquitectónico

El acceso al backend estará limitado a usuarios con credenciales y permitirá llevar a cabo las tareas de administración de contenido.

El frontend será de acceso público y permitirá visualizar el contenido generado desde el backend.

4.2.1 Diseño del servidor

Como ya se comentó en la sección 1, uno de los principios REST de mayor importancia es el de sistema por capas, el cual implica que un componente no puede ver más allá de la capa inmediata con la cual interactúa.

El diseño del servidor se basa en una arquitectura de tres capas con separación lógica de la capa de acceso a datos, la capa de de negocio y la capa de servicios

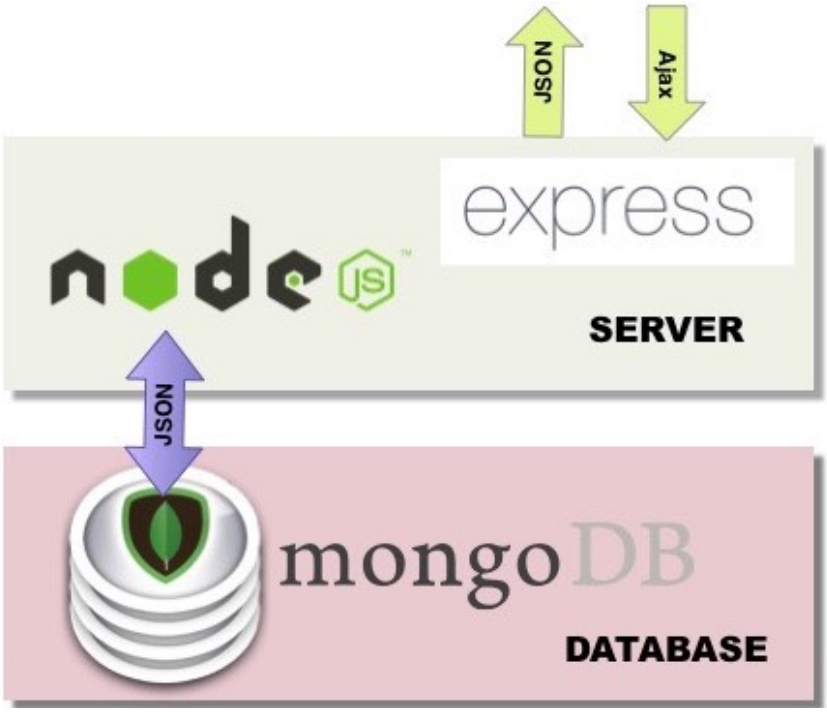


Gráfico 10: Diseño de servidor

Capa de presentación

Al restringir el conocimiento del sistema a una sola capa, se impone un límite en la complejidad del sistema en general, promoviendo así la independencia de los sustratos.

- Se comunica únicamente con la capa de negocio.
- Implementa una API que recoge las peticiones del cliente y genera los mensajes de respuesta o error correspondientes.
- La interacción entre el cliente y el servidor no tiene estado entre solicitudes. La solicitud del cliente al servidor debe contener toda la información necesaria para comprender la solicitud.
- El estado y la funcionalidad del sistema se dividen en recursos. Un recurso es un elemento de interés, una identidad conceptual que se expone a los clientes

- Cada recurso es de acceso único a través de una URI (Universal Resource Identifier – identificador de recursos universal).

- Todos los recursos comparten una interfaz uniforme para la transferencia de estados entre cliente y servidor. Se usan métodos estándar HTTP como GET, PUT, POST y DELETE. El motor del estado de la aplicación es Hypermedia y las representaciones de recursos se interconectan mediante hipervínculos.

Capa de negocio

La capa de negocio cumple la función de intermediario en el intercambio de datos entre la capa presentación o servicios y la capa de acceso a datos. Es aquí donde se realizan la mayor parte de las funciones de la aplicación, como procesamiento de datos, implementación de funciones de negocios, etc.

Se denomina capa de negocio o capa de lógica del negocio, porque es aquí donde se establecen todas las reglas que deben cumplirse

Capa de acceso a datos

La capa de acceso a datos contiene la lógica principal de acceso y persistencia de datos dentro de nuestra aplicación. La capa de acceso a datos tiene que soportar no solo el almacenamiento de datos, sino la recuperación de información de formas complejas y la concurrencia de múltiples usuarios accediendo a la información.

Los principales beneficios del uso de este tipo de arquitectura son:

- **Abstracción**. El uso de capas permite un alto nivel de abstracción ya que la comunicación entre capas queda bien definida mediante interfaces.

- **Aislamiento.** El estilo de arquitectura de capas permite aislar los cambios en tecnologías a ciertas capas para reducir el impacto en el sistema total.

- **Escalabilidad.**

4.2.2 Diseño del cliente (Interfaz gráfica de administración).

La interfaz de administración se ha desarrollado el patrón de diseño Modelo-Vista-Modelo de Vista (Model-View-ViewModel en inglés y abreviado como MVVM) haciendo uso del framework AngularJS.

EL MODELO.

El modelo representa los datos o la información con la que trabajamos. Un ejemplo de modelo de datos típico podría ser el que representa una cuenta de usuario (nombre, email, etc).

El modelo contiene la información pero no las acciones o servicios que la manipulan. La lógica de la aplicación o reglas de negocio son generalmente mantenidas en clases separadas del modelo que actúan sobre él. Como excepción el modelo puede contener la lógica necesaria para la validación de la información. Por ejemplo asegurándose que una dirección de correo electrónico cumple los requerimientos de una expresión regular.

LA VISTA

Es la única parte de la aplicación con la que el usuario final interactúa. Se trata de una interfaz interactiva que representa el estado del ViewModel. En este sentido se considera que una vista MVVM es activa.

Una vista pasiva no tiene conocimiento real del modelo de nuestra aplicación y es manipulada por el controlador. En MVVM la vista es activa. A diferencia de una vista pasiva sin conocimiento del modelo, y bajo el manejo total de un controlador o presentador, la vista en MVVM contiene comportamientos, eventos y enlaces a datos que, hasta cierto punto, necesita saber sobre el modelo subyacente y el Modelo de Vista.

El papel de la vista es representar la información para presentarla al usuario final. Por ejemplo una fecha puede estar representada en el modelo como el número de segundos contados desde la media noche del 1 de enero de 1970 (Fecha Linux) y ser presentado al usuario como día, mes y año en su zona horaria actual.

La vista no es responsable de gestionar su estado, es el modelo de vista quien se encarga de ello y mantiene al tanto a la vista de los cambios. Vista y Modelo de Vista deben estar perfectamente sincronizados.

La vista es la encargada de manejar sus propios eventos (presión de teclas, movimientos de ratón, gestos en pantalla táctil, etc) mapeándolos con el Modelo de Vista cuando sea necesario.

Las principales ventajas e inconvenientes del MVVM son

Ventajas

- Facilita el desarrollo en paralelo de la interfaz de usuario y de la lógica asociada.
- Mayor adaptabilidad a las pruebas unitarias.
- Posibilidad de reutilización de código
- El nivel de abstracción que ofrece la Vista nos permitirá escribir menos código, mas interpretable y comprensible

Desventajas

Para interfaces de usuario simples puede llegar a no ser justificado su uso.
Los enlaces de datos pueden provocar que el código sea más difícil de depurar.

EL MODELO DE VISTA (CONTROLADOR)

El Modelo de Vista se puede considerar como un controlador especializado que actúa como conversor de datos, transformando la información del Modelo en información para la Vista y pasando los comandos desde la Vista al Modelo.

Por ejemplo imaginemos que nuestro modelo contiene una fecha en formato unix (por ejemplo 1333832407) y que la Vista lo representa como (07/04/2012). Para que el Modelo no tenga que tener conocimiento sobre el formato presentado al usuario el Modelo de Vista actuará de intermediario para obtener el dato formateado de la vista y presentarlo al modelo en formato unix.

De esta forma el Modelo de Vista introduce el concepto de Separación de la Presentación, es decir, mantiene al modelo separado y protegido de los minuciosos detalles de la vista.

4.3 Diseño detallado

Una vez definido el diseño arquitectónico de la aplicación vamos a proceder a adaptar los requerimientos a la arquitectura dada. Para ello, el desarrollo del sistema comienza con el

diseño en detalle del servidor de la aplicación, el cual debe proporcionar la lógica adecuada para soportar las peticiones de datos de las aplicaciones cliente y los requerimientos de seguridad del sistema.

4.3.1 Diseño del servidor

Como vimos en el apartado anterior, la arquitectura del servidor se divide en tres capas lógicas que separan las distintas funcionalidades del sistema. A continuación se explicará en detalle como los requisitos del proyecto se integran en la arquitectura diseñada y en la infraestructura de las bases de datos.

En base a los requisitos del proyecto se han diseñado una serie de recursos que definen la API de llamadas del servidor, estos recursos engloban las distintas entidades que el usuario podrá consultar o modificar en la base de datos a través del servidor. Cada uno de los recursos, siguiendo las directrices de diseño de una arquitectura REST, serán visible por un patrón de URL definido, sobre el cual se podrán mandar desde la aplicación móvil peticiones para crear, consultar, modificar o borrar dependiendo de los requisitos dados.

Los recursos que componen la API y las operaciones de cada uno son los siguientes:

Recurso	URL	Peticiónes	Respuestas
Usuario	/users	GET	HTTP 200
	/users/{id}	GET POST PUT DELETE	HTTP 200 HTTP404 HTTP500
	/authenticate	POST	HTTP 200 HTTP404
Tipos de contenido	/fieldtypes	GET	HTTP 200
	fieldtypes/{categoryId}	GET POST PUT DELETE	HTTP 200 HTTP404
	/categories	GET	HTTP 200

Recurso	URL	Peticiónes	Respuestas
Categorías	/categories/{id}	GET POST PUT DELETE	HTTP 200 HTTP404 HTTP500
	/categories/{id}/content_items	GET	HTTP 200
Elementos de contenido	/content_items	GET	HTTP 200
	/content_items/{id}	GET POST PUT DELETE	HTTP 200 HTTP404 HTTP500
	/content_items/stats	GET	HTTP 200

Una vez definido los recursos que integrarán nuestra API, entraremos en detalle a definir los atributos presentes en cada recurso, los parámetros que necesitan y las posibles respuestas que entrega el servidor en cada caso.

SEGURIDAD

Uno de los aspectos más importantes en lo que se refiere a la seguridad del sistema es el sistema de autenticación empleado. Para nuestro sistema hemos optado por una autenticación basada en Tokens.

Antes de entrar en detalle de cómo funciona la autenticación basada en tokens vamos a repasar los mecanismos de autenticación tradicionales en los que las aplicaciones recuerdan quienes somos es almacenando la información de las sesiones de usuario en el servidor.

A continuación se muestra un gráfico de la autenticación basada en servidor.

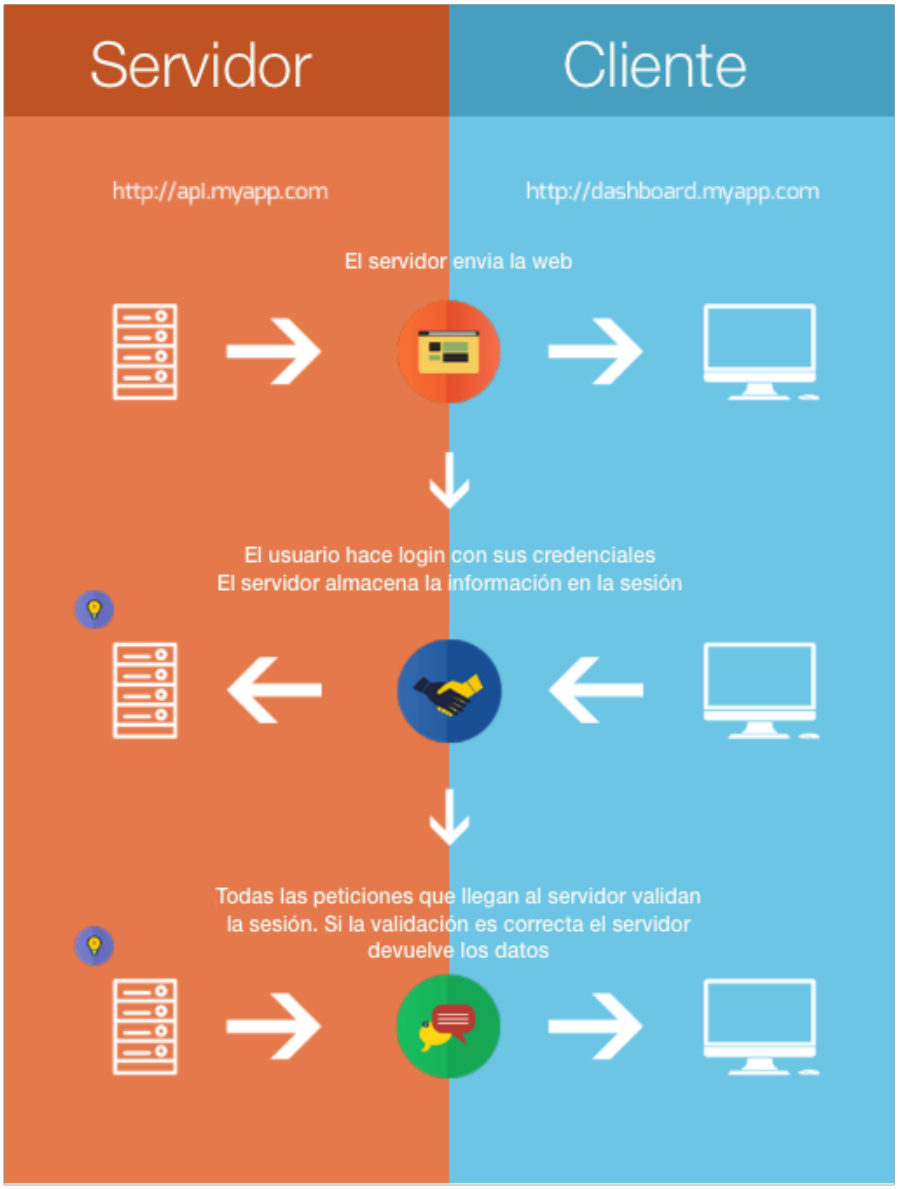


Gráfico 11: Autenticación basada en servidor

El auge de las aplicaciones basadas en web y la aparición de las aplicaciones móviles han puesto de manifiesto algunos problemas de este tipo de autenticación, sobre todo en lo que se refiere a la escalabilidad.

- **Sesiones**. El servidor debe mantener un registro de cada usuario que se registra en el sistema. Normalmente las sesiones se almacenan en memoria por lo que los sistemas con muchos usuarios autenticados consumirá muchos recursos.

- **Escalabilidad**. El hecho de mantener la información de sesión en memoria limita la posibilidad de escalar los recursos haciendo uso de balanceadores de carga.

- **CORS.** Si queremos que nuestra aplicación sea accedida desde diferentes tipos de dispositivos móviles tendremos que tener en cuenta posibles problemas de cross-origin (CORS). Cuando hacemos uso de llamadas AJAX para obtener recursos de otro dominio podemos enfrentarnos a problemas de peticiones denegadas.

Una vez vistas las limitaciones de la autenticación basada en sesiones del servidor vamos a pasar a explicar como funciona la autenticación basada en tokens.

La autenticación basada en tokens no almacena información acerca de los usuarios autenticados en el sistema, con lo que se resuelve la mayoría de los problemas mencionados anteriormente. A grandes rasgos el flujo de la autenticación basada en tokens es:

- El usuario solicita acceso haciendo uso de credenciales
- El servidor valida las credenciales y devuelve un token al usuario
- El cliente almacena el token y lo envía con cada petición que realice al servidor
- El servidor verifica el token y responde con los datos.

Cada petición al servidor debe incluir el token, que debe ser añadido en la cabecera HTTP. Para evitar problemas de CORS debemos habilitar el servidor para que acepte peticiones de cualquier dominio.

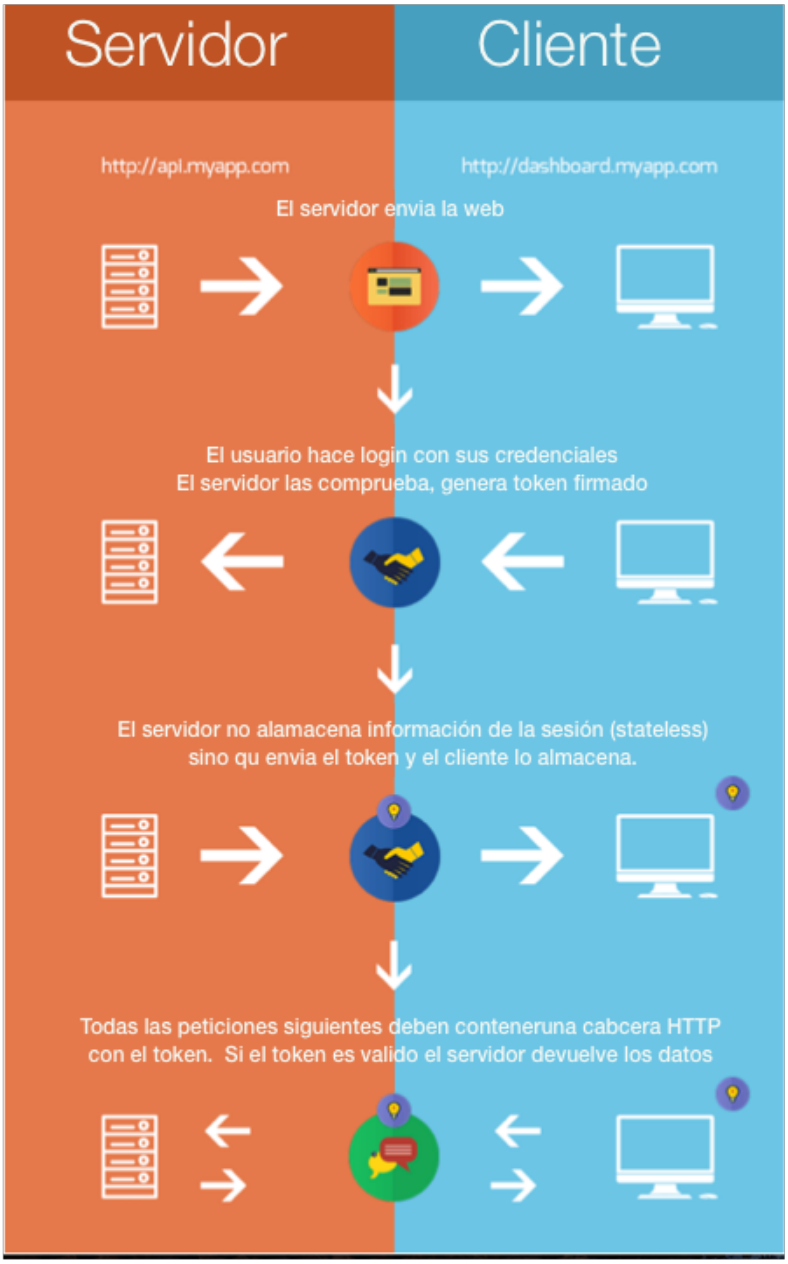


Gráfico 12: Autenticación basada en tokens

La principales ventajas del uso de tokens para la autenticación son:

Escalabilidad. Con esta nueva aproximación los tokens se almacenan en el cliente lo que deja la puerta abierta para la escalabilidad, permitiendo que los balanceadores de carga deriven a los usuarios a cualquiera de los servidores disponibles.

Seguridad. En este proceso no intervienen las cookies por lo que ayuda a solucionar los ataques CSRF (Cross-site request forgery). Incluso si el token se almacena en una cookie, la cookie sería simplemente un mecanismo de almacenaje y no un mecanismo de autenticación propiamente dicho.

Ampliación. Los tokens permiten construir aplicaciones que comparten permisos con otras aplicaciones. Por ejemplo podemos permitir que una aplicación haga uso de la api de twitter para importar los últimos tweets. Los tokens son un buen mecanismo para autorizar el acceso desde aplicaciones desarrolladas por otros.

Muti plataforma y multi dominio. Ya hemos hablado anteriormente de CORS y como los tokens solucionan este problema, por tanto nuestros datos estarán disponibles para peticiones desde cualquier dominio o dispositivo siempre que el usuario disponga de un token válido.

Nuestro sistema tendrá una serie de recursos públicos y otra serie de recursos privados que sólo podrán ser accedido por los usuarios que se hayan identificado en el sistema haciendo uso de credenciales y que dispongan de un token válido.

MODELO DE DATOS

Para el modelado de datos se ha optado por una base de datos noSQL. La principal razón para esta elección es la flexibilidad que ofrece frente a las bases de datos relacionales en las que las tablas deben estar normalizadas y se debe definir la relación existente entre las distintas tablas.

Dentro de los tipos de bases de datos noSQL hemos optado por una base de datos de documentos. En este tipo de base de datos el principal concepto es el documento que puede estar almacenado en formato XML, JSON, BSON. Los documentos almacenados serán similares pero pudiendo existir diferencias entre ellos. Hemos usado el concepto de agregación en vez del de normalización. Un documento se puede entender como un conjunto de entidades de bases de datos agrupados como una gran entidad.

En nuestro caso los elementos de contenido tendrán una serie de campos comunes y una serie de componentes que variarán en función de la categoría a la que estén asignados.

Podríamos aplicar un modelo relacional normalizado con una tabla independiente por cada categoría. Pero usando este esquema cada vez que queramos ampliar el sistema con nuevas categorías nos veremos obligados a actualizar el modelo de datos con nuevas tablas.

Uno de los requisitos del proyecto es que los usuarios puedan crear sus propias categorías definiendo los tipos de contenido que la componen, por tanto la opción de usar una base de datos relacional no parece la más elegante. Aprovechando la flexibilidad que nos ofrece las bases de datos noSQL podemos representar todos los elementos de contenido haciendo uso de una única agregación (content_item).

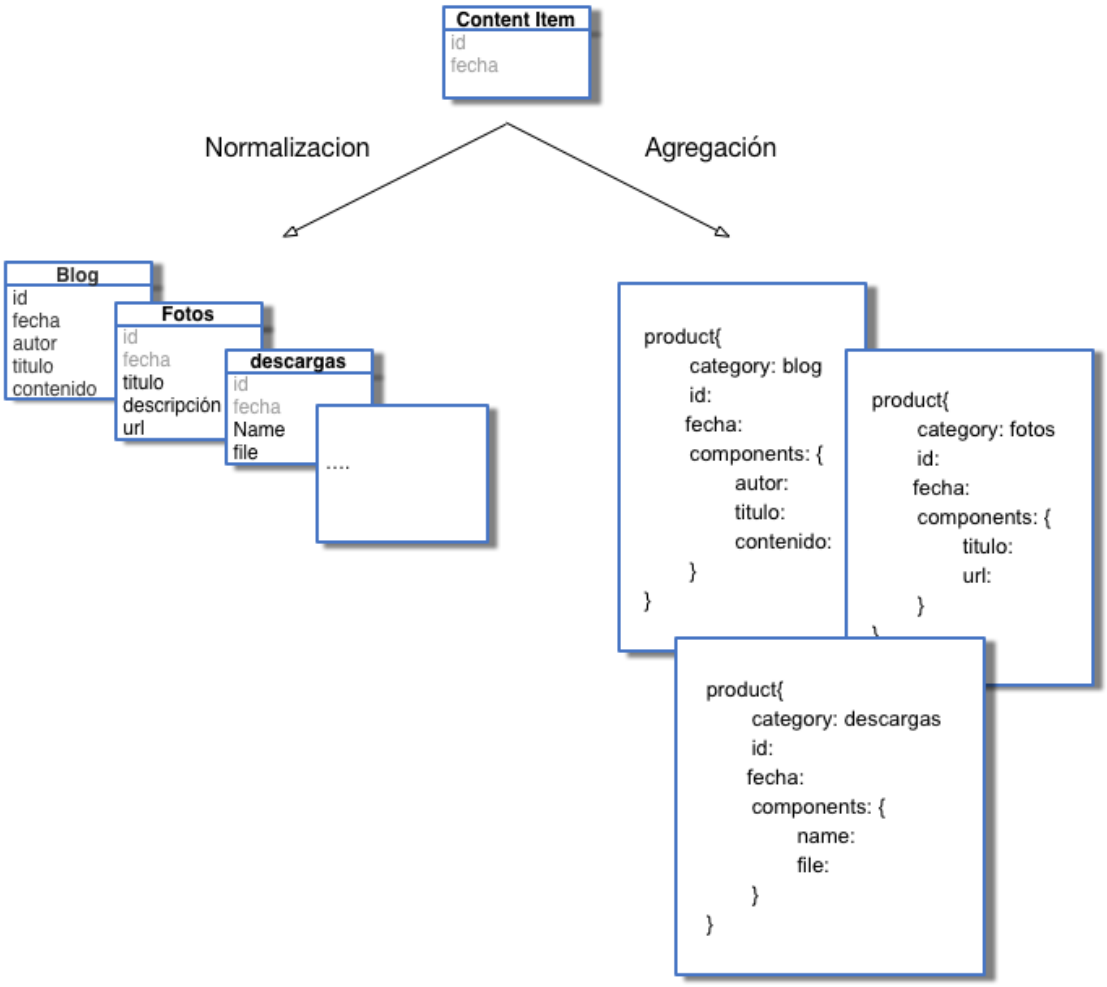


Gráfico 13: modelo de datos.Normalización vs Agregación

Usuario

El recurso usuario corresponde al usuario de la aplicación. Se considera usuario de la aplicación a cualquier persona con las credenciales para acceder a la interfaz de administración.

El modelo de datos del usuario se ha definido en el sistema con los siguientes campos

Gráfico 14: modelo de datos del usuario

Como se detalla en la sección anterior el sistema hace uso de un modelo de autenticación basada en tokens. El cliente se identifica en el sistema enviando su email y su contraseña y el servidor responde con un token. El cliente debe almacenar el token y enviarlo en cada petición. El servidor validará el token y si es correcto devolverá la información necesaria.

Para obtener el token de validación es necesario hacer una petición **HTTP POST a / authenticate** incluyendo en el cuerpo de la petición un json con el email y la contraseña del usuario:

```
{
        "email":"jhondoe@email.com",
        "password":"123456"
}
```

Si se el usuario se encuentra en la base de datos y las credenciales son correctas el servidor responderá con OK HTTP 200 incluyendo en el body, el token, la fecha de caducidad del token y la información del usuario.

```
{
        "token":"eyJ0eXAiOiJKV1QiLCJh...",
        "expires": 1480499168851,
        "user": {
                "_id": "55d327238227295c17780ba5",
                "password":",
                "name": "jhon doe",
                "email": "jhon@doe.com",
                "__v": 0,
                "admin": true
    }
}
```


En caso de que el usuario no se encuentre en la base de datos o las credenciales no sean correctas el servidor responderá con un error HTTP 401 (No autorizado).

Para obtener la lista de los usuarios dados de alta en el sistema se debe realizar una petición HTTP **GET a /users**. Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si la validación del token es correcta devolverá HTTP 200 y en el body un array con la información de los usuarios disponibles en el sistema y en caso contrario devolverá un error HTTP 401.


```
[
 {
  "_id": "55d327238227295c17780ba5",
  "name": "jhon doe",
  "email": "jhon@miemail.com",
  "__v": 0,
  "admin": true
 },
 {
  "_id": "5825c7daa024432d94c0760b",
  "name": "test1",
  "email": "test@test.com",
  "__v": 0,
  "admin": true
 }
]
```


Para obtener los datos de un usuario concreto se debe hacer una petición **HTTP GET /users/{id}** donde id es el identificador del usuario en el sistema. Al tratarse de un recurso

privado deberá incluir una cabecera HTTP x-access-token con un token válido. Si no encuentra al usuario el servidor responderá con un error HTTP 404 (no encontrado). Si el id se corresponde con un usuario del sistema y el token es válido el sistema responderá con un HTTP 200 devolviendo la información del usuario en el cuerpo del mensaje.


```
{
    "_id": "55d327238227295c17780ba5",
     "name": "Javier Rojano",
    "email": “jhon@doe.com",
    "__v": 0,
    "admin": true
}
```


La creación de usuarios se realiza mediante peticiones **HTTP POST a /users** incluyendo en el body un json con los datos del usuario:


```
{
   "password": "xxxx",
   "name": “Jhon doe",
   "email": "xxxx@mail.com",
   "admin": true
 }
```


Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si el usuario ha sido creado y las credenciales son correctas el servidor responderá con OK HTTP 200 incluyendo en el body, el token, la fecha de caducidad del token y la información del usuario.


```
{
   “token”:”eyJ0eXAiOiJKV1QiLCJhb...",
   "expires": 1480499168851,
   "user": {
   "_id": "55d327238227295c17780ba5",
   "name": "Javier Rojano",
   "email": "test@gmail.com",
   "__v": 0,
   "admin": true
  }
}
```

En caso de error al crear el usuario o que las credenciales no sean correctas el servidor responderá con un error HTTP 500 o error HTTP 401 respectivamente.

La modificación de los datos de un usuario se realiza mediante peticiones **HTTP PUT a / users/{id}** donde id es el identificador del usuario que queremos modificar. En el body de la petición deberemos incluir un json con los datos del usuario que deseemos modificar. Por ejemplo si quisiéramos modificar el nombre y el email:

```
{
  "name": "Javier Rojano",
  "email": "xxxx@mail.com"
 }
```

Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si se el sistema encuentra al usuario en la base de datos y las credenciales son correctas el servidor responderá con OK HTTP 200 incluyendo en el body, el token, la fecha de caducidad del token y la información del usuario actualizada.

```
{
   “token”:”eyJ0eXAiOiJKV1QiLCJhbGciOiJI...",
   "expires": 1480499168851,
   "user": {
   "_id": "55d327238227295c17780ba5",
   "name": "Javier Rojano",
   "email": "miemail@gmail.com",
   "__v": 0,
   "admin": true
  }
}
```

En caso de que el usuario no se encuentre en la base de datos o las credenciales no sean correctas el servidor responderá con un error HTTP 500 o error HTTP 401 respectivamente.

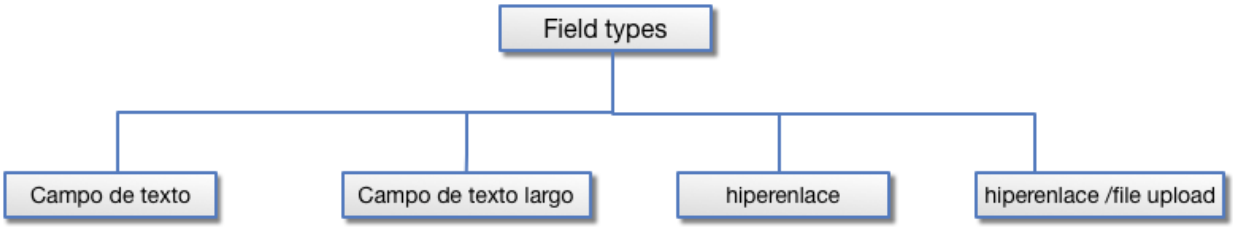
Para eliminar a un usuario del sistema se debe realizar una petición **HTTP DELETE a /users/{id}** donde id es el identificador del usuario que queremos eliminar. Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si el sistema encuentra al usuario y lo borra responderá con un HTTP 200 mientras que si no encuentra el usuario o no puede borrarlo devolverá un HTTP 500.

TIPO DE CONTENIDO

El recurso tipo de contenido (field types) representa los elementos de contenido que conforman una categoría. Los tipos de contenidos están predefinidos en el sistema y son:

- **Campo de texto**. Se trata de un campo de texto con longitud máxima de 255 caracteres.
- **Campo de texto largo**. Campo de texto de sin limitación de longitud (salvo la impuesta por el sistema de base de datos). Puede contener texto enriquecido en formato html.
- **Hiperenlace**. Almacena la url de un enlace hipermedia externo.
- **Hiperenlace o fileUpload**. Almacena la url de un enlace hipermedia externo o un archivo que ha sido subido al servidor por los usuarios del sistema.




```
{
    "_id" : ObjectId("54478f5aa7ce1cc553abed20"),
    "name" : "cms_text_input",
    "desc" : "Campo de texto",
    "opts" : [
        {
            "maxlength" : "255"
        }
    ],

}
```

Gráfico 15: Tipos de contenido

Para obtener la lista completa de los tipos de contenido definidos en el sistema debe realizar una petición **HTTP GET a /fieldtypes**. El servidor responderá con un HTTP 200 y un array de los tipos de contenido.


```
[
 {
  "_id": "54478f5aa7ce1cc553abed20",
  "name": "cms_text_input",
  "desc": "Campo de texto",
  "__v": 0,
  "opts": [
   {
    "maxlength": "255"
   }
  ]
 },
 {
  "_id": "5447b5d0a7ce1cc553abed24",
  "name": "cms_textarea",
```

```
  "desc": "Campo de texto largo",
  "__v": 0,
  "opts": [
   {
   "richText": true
   }
  ]
 },
 {
  "_id": "544ac2d3f0f81f15a892c93c",
  "name": "cms_link",
  "desc": "hiperenlace.",
  "opts": []
 },
 {
  "_id": "54da55b178dc0e5fc69dd1c8",
  "name": "cms-linkfile",
  "desc": "enlace a una url o fileUpload",
  "opts": [
   {
    "type": "TAGS INPUT",
    "options": [
     {
      "text": "application/pdf"
     },
     {
      "text": "image/png"
     },
     {
      "text": "image/jpeg"
     },
     {
      "text": "application/msword"
     },
     {
      "text": "image/gif"
     },
     {
      "text": "audio/mpeg3"
     }
    ],
    "label": "Accepted Mime Types",
    "desc": "Mime Types",
    "name": "mime_types"
```

```
    }
   ]
  }
 ]
```

Para obtener una lista de los tipos de contenido de una determinada categoría se debe realizar una petición **HTTP GET a /fieldtypes/{catgory_id}** donde category_id es el identificador de la categoría de la que queremos obtener los tipos de campo.

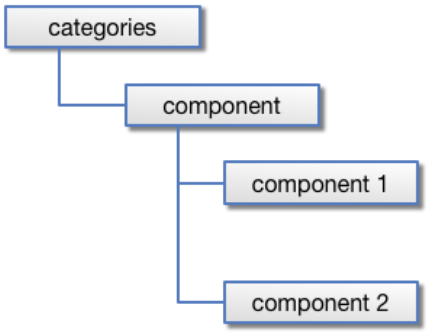
El servidor responderá con un HTTP 200 y un array de los tipos de contenido.

CATEGORÍAS

Todo el contenido del sistema está asociado a una categoría. La categoría además de agrupar los diferentes tipos de contenido del sistema sirve también para definir que elementos de información contendrán.

Pongamos un ejemplo. Supongamos que queremos desarrollar un sitio web en el que se van a publicar noticias, una lista de recursos para descargar y cierta información corporativa. Para ello crearemos 3 categorías en el sistema:

- **Información corporativa**. compuesta por un título (campo de texto) y contenido (campo de texto largo).
- **Noticias**. Compuesta por un título (campo de texto), una descripción (campo de texto largo), una foto (Hiperenlace o fileUpload) y el contenido (campo de texto largo)
- **Descargas**. Compuesta por nombre (campo de texto), descripción (campo de texto largo) y enlace (Hiperenlace o fileUpload).




```
{
    "_id" : ObjectId("5447b5fca7ce1cc553abed25"),
    "desc" : "Contenido de distintas secciones de la web.",
    "items" : NumberInt(12),
     "name" : "Contenido Principal",
    "canDelete" : true,
    "canAdd" : true,
    "components" : [
        {
            "name" : "title",
            "desc" : "Título de la sección",
            "type_id" : ObjectId("54478f5aa7ce1cc553abed20"),
            "_id" : ObjectId("5447b5fca7ce1cc553abed27"),
            "label" : "Título",
            "show" : true
        },
        {
            "name" : "content",
            "desc" : "Contenido de la sección",
            "type_id" : ObjectId("5447b5d0a7ce1cc553abed24"),
            "_id" : ObjectId("5447b5fca7ce1cc553abed26"),
            "label" : "Contenido",
            "show" : false
        }
    ]

}
```

Gráfico 16: Ejemplo de categorías

Todas las categorías tienen una serie de campos comunes:

- **Name**: Nombre de la categoría.
- **Desc**: descripción de la categoría.
- **items**: número de elementos asignados a esta categoría.
- **CanDelete**: indica si se permite a los usuarios eliminar contenido de estas categorías. En determinadas ocasiones nos puede interesar impedir que los usuarios puedan borrar o añadir contenido de ciertas categorías. Un ejemplo podría ser el contenido estático de una web. Podemos crear una categoría que contenga un campo de texto para el título y un campo de texto largo para el contenido. Imaginemos que creamos dos secciones, inicio y quienes somos. El cliente solicita a través de la API REST estas secciones a través de su id único, si permitimos que alguien borre algunas de estas secciones, el cliente no será capaz de obtenerla.
- **CanAdd**: indica si se permite a los usuarios añadir contenido de estas categorías.

A parte de los campos comunes el objeto categoría tiene un array con los tipos de contenido que la componen. El campo Type_id relaciona el componente con el tipo de contenido correspondiente.

Para obtener la información de una categoría concreta se debe realizar una petición HTTP GET a **/categories/{id}** donde id es el identificador de la categoría. El servidor responderá con un HTTP 200 y la información de la categoría en el body.


```
{
	"_id": "5447b5fca7ce1cc553abed25",
	"desc": "Contenido de distintas secciones de la web.",
	"__v": 0,
	"name": "Contenido Principal",
	"CanAdd": false,
	"components": [{
		"name": "title",
		"desc": "Título de la sección",
		"type_id": "54478f5aa7ce1cc553abed20",
		"_id": "5447b5fca7ce1cc553abed27",
		"label": "Título",
		"show": true
	}, {
		"name": "content",
		"desc": "Contenido de la sección",
		"type_id": "5447b5d0a7ce1cc553abed24",
		"_id": "5447b5fca7ce1cc553abed26",
		"label": "Contenido",
		"show": false
	}],
	"items": 12,
	"canAdd": true,
	"canDelete": true
}
```


Para obtener la lista completa de categorías debe realizar una petición **HTTP GET a / categories.** El servidor responderá con un HTTP 200 y un array con la lista de categorías.

```
[
  {
    "_id": "5447b5fca7ce1cc553abed25",
    "desc": "Contenido de distintas secciones de la web.",
    "__v": 0,
    "name": "Contenido Principal",
    "CanAdd": false,
    "components": [
      {
        "label": "Título",
        "_id": "5447b5fca7ce1cc553abed27",
        "type_id": {
          "_id": "54478f5aa7ce1cc553abed20",
          "name": "cms_text_input",
          "desc": "Campo de texto",
          "__v": 0,
          "opts": [
            {
              "maxlength": "255"
            }
          ]
        },
        "desc": "Título de la sección",
        "name": "title",
        "show": true
      },
      {
        "label": "Contenido",
        "_id": "5447b5fca7ce1cc553abed26",
        "type_id": {
          "_id": "5447b5d0a7ce1cc553abed24",
          "name": "cms_textarea",
          "desc": "Campo de texto largo",
          "__v": 0,
          "opts": [
            {
              "richText": true
            }
          ]
        },
        "desc": "Contenido de la sección",
        "name": "content",
        "show": false
      }
```

```
  ],
  "items": 12,
  "canAdd": true,
  "canDelete": true
},
{
  "_id": "544acd4b36da7fa145a58104",
  "name": "Publicaciones y blog",
  "desc": "Publicaciones y blog",
  "__v": 0,
  "components": [
   {
    "name": "urlFile",
    "label": "url",
    "desc": "Url del enlace",
    "type_id": {
     "_id": "54da55b178dc0e5fc69dd1c8",
     "name": "cms-linkfile",
     "desc": "enlace a una url o fileUpload",
     "opts": [
      {
       "type": "TAGS INPUT",
       "options": [
        {
         "text": "application/pdf"
        },
        {
         "text": "image/png"
        },
        {
         "text": "image/jpeg"
        },
        {
         "text": "application/msword"
        },
        {
         "text": "image/gif"
        },
        {
         "text": "audio/mpeg3"
        }
       ],
       "label": "Accepted Mime Types",
       "desc": "Mime Types",
```

```
            "name": "mime_types"
          }
        ]
      },
      "_id": "544acd4b36da7fa145a58106",
      "show": true
    },
    {
      "name": "texto",
      "label": "texto",
      "desc": "Texto del enlace",
      "type_id": {
        "_id": "5447b5d0a7ce1cc553abed24",
        "name": "cms_textarea",
        "desc": "Campo de texto largo",
        "__v": 0,
        "opts": [
          {
            "richText": true
          }
        ]
      },
      "_id": "544acd4b36da7fa145a58105",
      "show": true
    }
  ],
  "items": 10,
  "canAdd": true,
  "canDelete": true
 }
]
```


La creación de categorías se realiza mediante peticiones HTTP **POST a /categories** incluyendo en el body un json con los datos para la nueva categoría:


```
{
  "desc": "Contenido de distintas secciones de la web.",
  "components": [
    {
      "name": "title",
      "desc": "Título de la sección",
      "type_id": "54478f5aa7ce1cc553abed20",
      "label": "Título",
```

```
            "show": true
        },
        {
            "name": "content",
            "desc": "Contenido de la sección",
            "type_id": "5447b5d0a7ce1cc553abed24",
            "label": "Contenido",
            "show": false
        }
    ],
    "name": "Test",
    "canDelete": false,
    "canAdd": false
}
```


Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si el sistema ha podido crear la categoría y las credenciales son correctas el servidor responderá con OK HTTP 200 incluyendo en el body los datos de la categoría recién creada.

En caso de error o las credenciales no sean correctas el servidor responderá con un error HTTP 500 o error HTTP 401 respectivamente.

La modificación de los datos de una categoría se realiza mediante peticiones **HTTP POST a /categories/{id}** donde id es el identificador de la categoría que queremos modificar. En el body de la petición deberemos incluir un json con los datos a modificar en la categoría.

Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si se el sistema encuentra la categoría en la base de datos y las credenciales son correctas el servidor responderá con OK HTTP 200 incluyendo en el body los datos de la categoría modificada.

En caso de que la categoría no se encuentre en la base de datos o las credenciales no sean correctas el servidor responderá con un error HTTP 500 o error HTTP 401 respectivamente.

Para eliminar una categoría del sistema se debe realizar una petición **HTTP DELETE a /categories/{id}** donde id es el identificador de la categoría que queremos eliminar. Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si el sistema encuentra la categoría y la borra responderá con un HTTP 200 mientras que si no la encuentra o no puede borrarla devolverá un HTTP 500.

ELEMENTOS DE CONTENIDO

El recurso contenido constituye la unidad de información principal del sistema. Como ya comentamos anteriormente los elementos de contenido deben pertenecer a una categoría que determinará los tipos de contenido que contiene. Como el sistema debe ser multiidioma los elementos de contenido contendrán un array con una entrada por cada idioma. Cada entrada del array idiomas representará los tipos de contenido en el idioma correspondiente.

A parte de la categoría a la que pertenece y la información del contenido en los diferentes idiomas el recurso elemento de contenido almacenará información sobre la fecha de creación y de la última modificación.

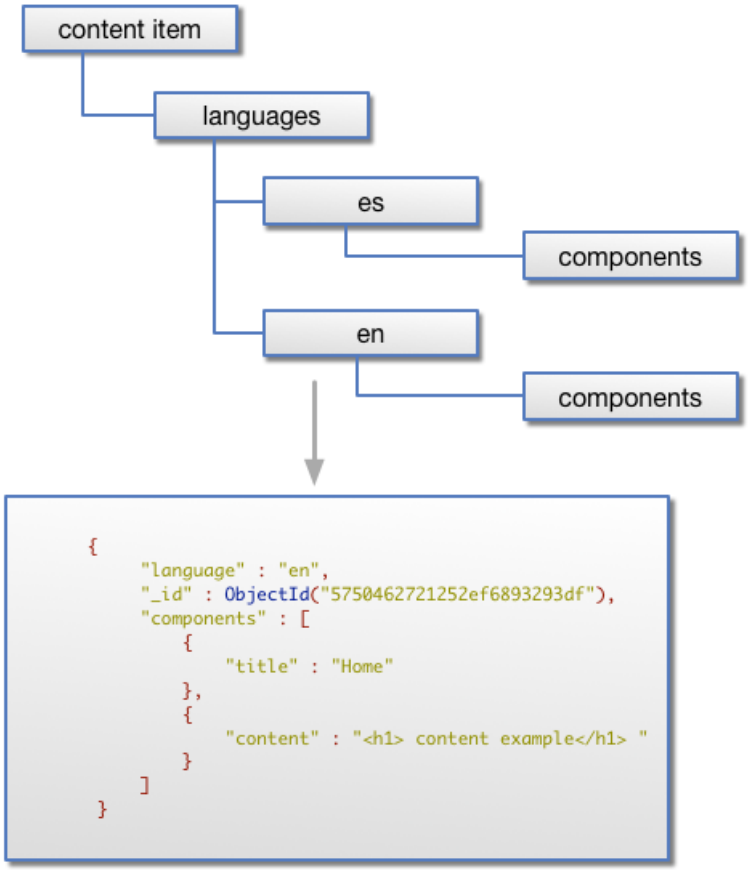


Gráfico 17: Elemento de contenido.

Para obtener la información de un elemento de contenido específico se debe realizar una petición HTTP **GET a /content_items/{id}** donde id es el identificador del elemento contenido. El servidor responderá con un HTTP 200 y la información del contenido en el body.


```
{
 "_id": "5750462721252ef6893293de",
 "category_id": "5447b5fca7ce1cc553abed25",
 "__v": 3,
 "languages": [
  {
   "language": "en",
   "_id": "5750462721252ef6893293df",
   "components": [
    {
     "title": "Home"
    },
```

```
      {
        "content": "contentet example"
      }
    ],
    "id": "5750462721252ef6893293df"
  },
  {
    "language": "es",
    "_id": "5750480d21252ef6893293e0",
    "components": [
      {
        "title": "Inicio"
      },
      {
        "content": "contenido de ejemplo"
      }
    ],
    "id": "5750480d21252ef6893293e0"
  }
 ],
 "updated": "2016-06-02T14:43:51.935Z",
 "created": "2016-06-02T14:43:51.935Z",
 "en": [
  {
    "title": "Home"
  },
  {
    "content": "<h1 style=\"color: rgb(255, 255, 255);text-align: left;background-color: rgb(60, 60,
59);\”>«…»</h1> "
  }
 ],
 "es": [
  {
    "title": "Inicio"
  },
  {
    "content": "<h1 style=\"color: rgb(255, 255, 255);text-align: left;background-color: rgb(60, 60,
59);\”>«…»</h1>”
  }
 ],
 "id": "5750462721252ef6893293de"
}
```

Para obtener la lista completa de elementos de contenido debe realizar una petición **HTTP GET a /content_items**. El servidor responderá con un HTTP 200 y un array con la lista de elementos de contenido.

La creación de elementos de contenido se realiza mediante peticiones **HTTP POST a /content_items** incluyendo en el body un json con los datos para el nuevo contenido. Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si el sistema ha podido crear el contenido y las credenciales son correctas el servidor responderá con OK HTTP 200 incluyendo en el body los datos del contenido recién creado. En caso de error o si las credenciales son erróneas el servidor responderá con un error HTTP 500 o error HTTP 401 respectivamente.

La modificación del contenido se realiza mediante peticiones **HTTP PUT a /content_items/{id}** donde id es el identificador del contenido que queremos modificar. En el body de la petición deberemos incluir un json con los datos a modificar en el contenido. Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si el sistema encuentra el contenido en la base de datos y las credenciales son correctas el servidor responderá con OK HTTP 200 incluyendo en el body los datos del contenido modificado.

En caso de que el contenido no se encuentre en la base de datos o las credenciales no sean correctas el servidor responderá con un error HTTP 500 o error HTTP 401 respectivamente.

Para eliminar contenido se debe realizar una petición **HTTP DELETE a /categories/{id}** donde id es el identificador de la categoría que queremos eliminar. Al tratarse de un recurso privado deberá incluir una cabecera HTTP x-access-token con un token válido.

Si el sistema encuentra y la borra el contenido responderá con un HTTP 200 mientras que si no lo encuentra o no puede borrarlo devolverá un HTTP 500.

El siguiente gráfico representa las relaciones entre los distintos documentos de nuestro modelo de datos.

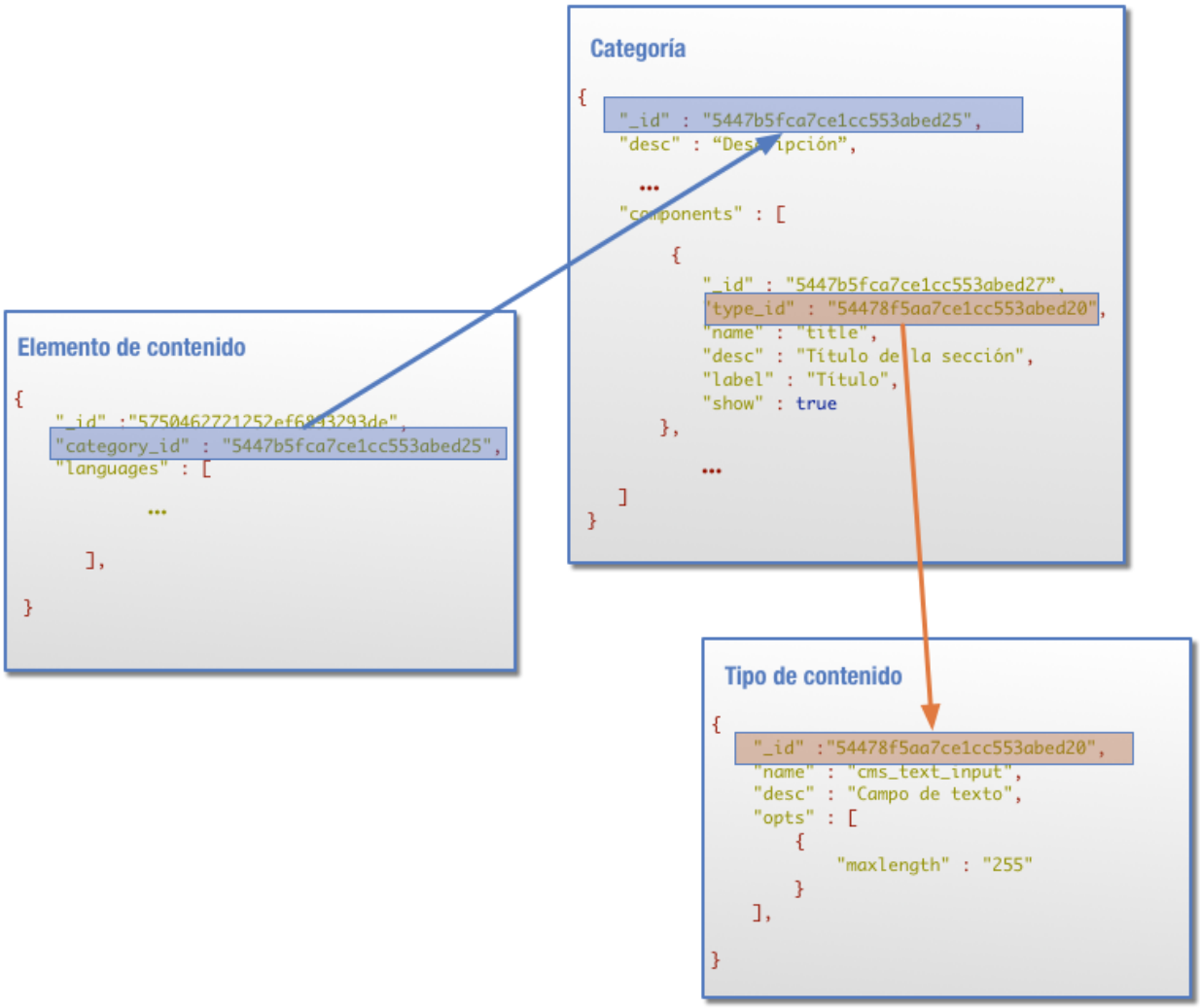


Gráfico 18: Relaciones del modelo de datos

Todo elemento de contenido tiene que estar asociado a una única categoría, y las categorías están compuestas por uno o varios tipos de contenidos.

4.3.2 Diseño del cliente (Interfaz gráfica de administración)

Para el diseño de la interfaz gráfica, en adelante nos referiremos a ella como panel de administración, nos hemos basado en la plantilla slim. Slim es una plantilla angularJS para paneles de control.

Slim tiene tres secciones principales:

- **Barra de herramientas.** En la barra de herramientas dispondremos de una serie de herramientas para personalizar la interfaz, cerrar la sesión o cambiar de idioma

- En la **barra lateral** se muestra el logotipo del CMS, y un menú desplegable con enlaces a todas las secciones del panel de control sobre las que el usuario tiene permiso. El contenido del menú variará en función de si se trata de un usuario administrador o de un usuario normal.

- **Área de contenido**. Es la sección principal de la interfaz en donde se presentará al usuario la información relativa a la sección activa en la barra lateral.

Si no tenemos una sesión abierta, la primera pantalla que se mostrará al usuario es la pantalla de inicio de sesión o login. En esta pantalla se solicita al usuario que indique su dirección de email y su contraseña.

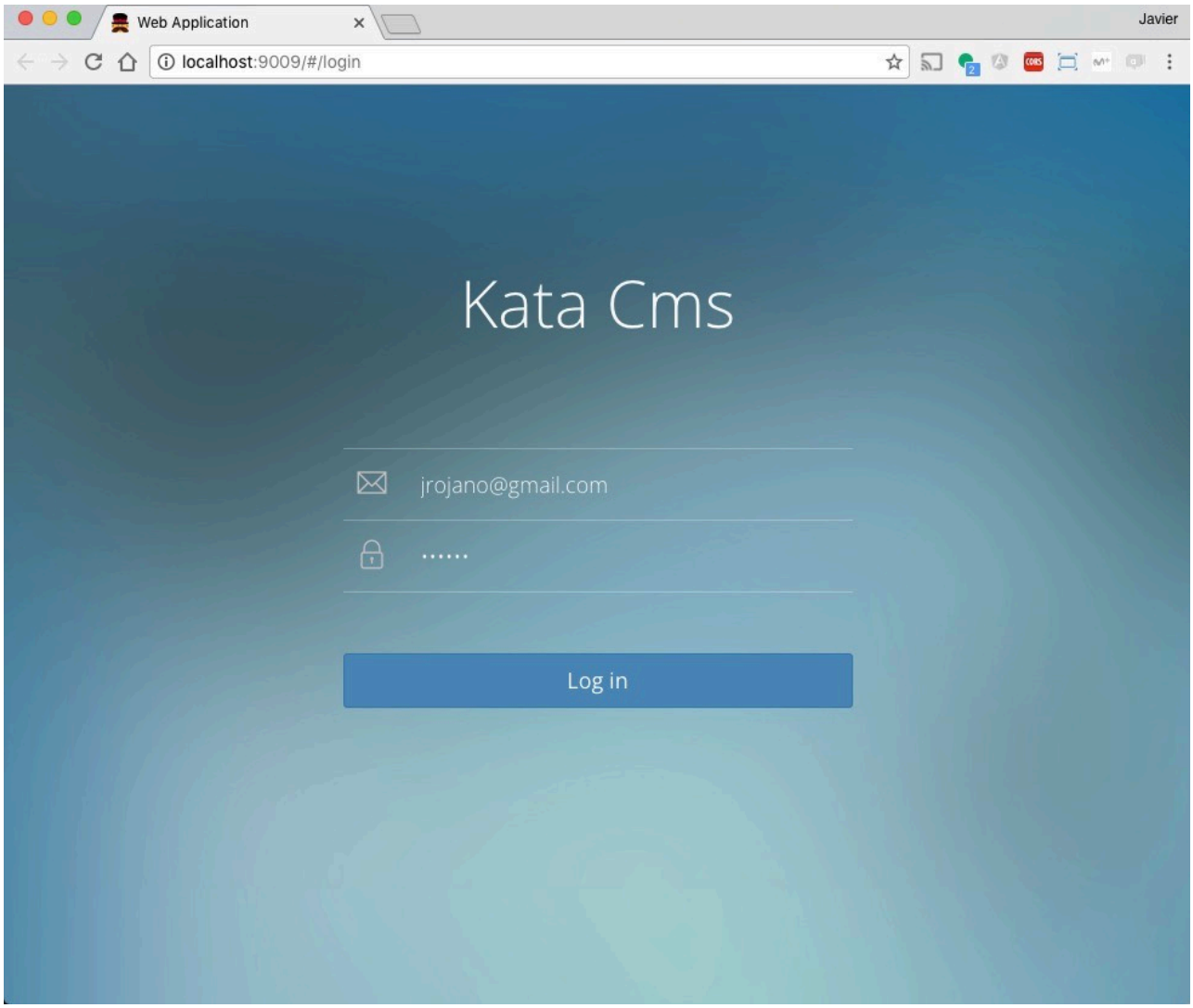


Gráfico 19: Pantalla login.

Si las credenciales son correctas si las credenciales son correctas el cliente almacenará el token y dirgirá al usuario al panel de administración. Si las credenciales no son correctas se mostrará un error.

Tanto los errores como los mensajes de estado se presentan al usuario utilizando un sistema de notificaciones flotantes que se muestran en la esquina inferior izquierda.

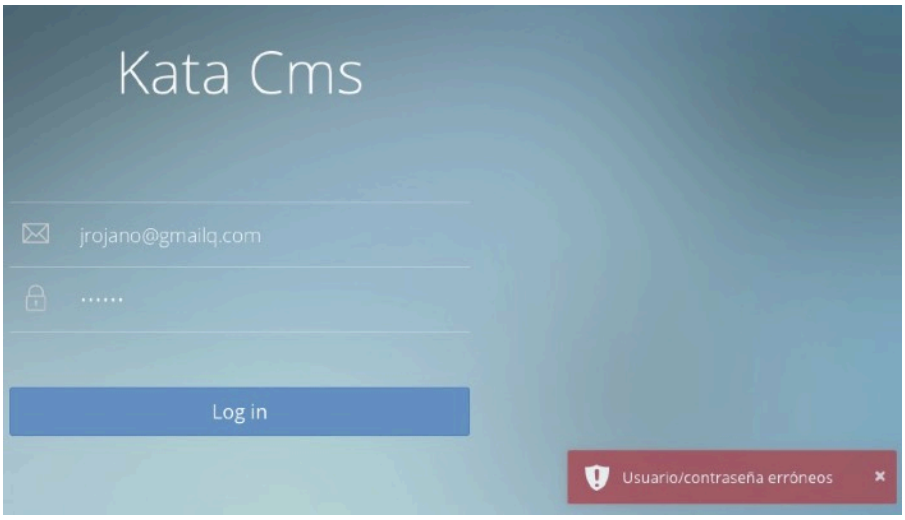


Gráfico 20: mensajes de error.

Una vez dentro del panel de administración, la pantalla de inicio muestra una serie de datos estadísticos del sistema incluyendo el número de categorías, el número de artículos (elementos de contenido), número de usuarios y una gráfica resumen.

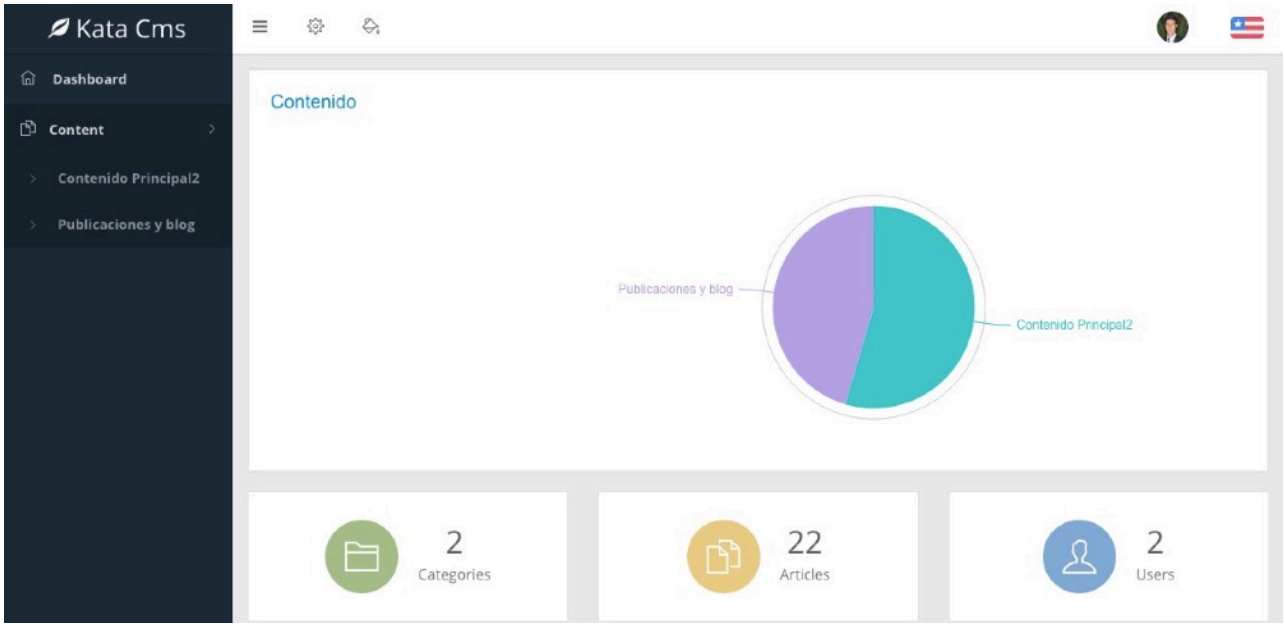


Gráfico 21: Pantalla de inicio.

Como ya se comentó anteriormente el contenido del menú lateral variará en función del rol del usuario. Si se trata de un usuario normal le aparecerán las opciones para crear contenido en cada una de las categorías definidas y un enlace a la página de inicio o dashboard. Si se trata de un usuario administrador a las opciones anteriores se añadirán la opción de gestionar las categorías y usuarios.

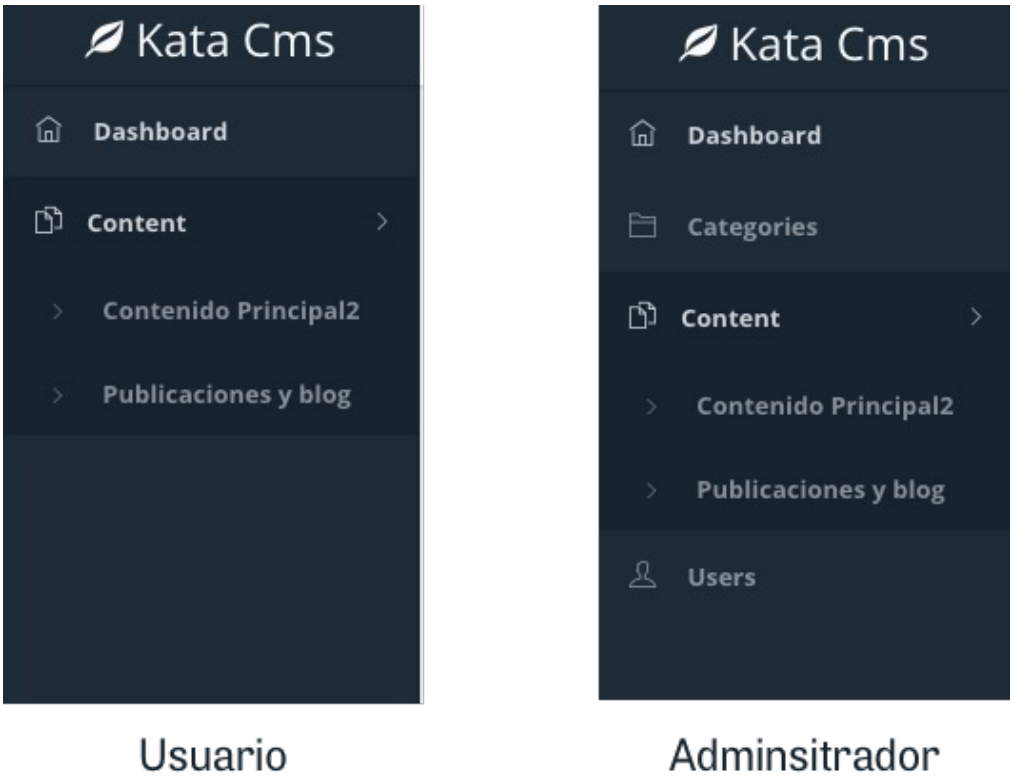


Gráfico 22: Barra lateral

ADMINISTRACIÓN DE CATEGORÍAS

Gráfico 23: lista de categorias

La pantalla de administración de categorías muestra un listado de todas las categorías disponibles en el sistema. El listado incluye en la zona superior un filtro para poder filtrar el contenido y un botón para poder añadir nuevas categorías.

Cada elemento del listado incluye el nombre y la descripción de la categoría y los botones para editar o borrar.

En el pie del listado se incluyen las opciones para la paginación.

El botón para crear categoría nos desplazará al formulario de creación de categoría.

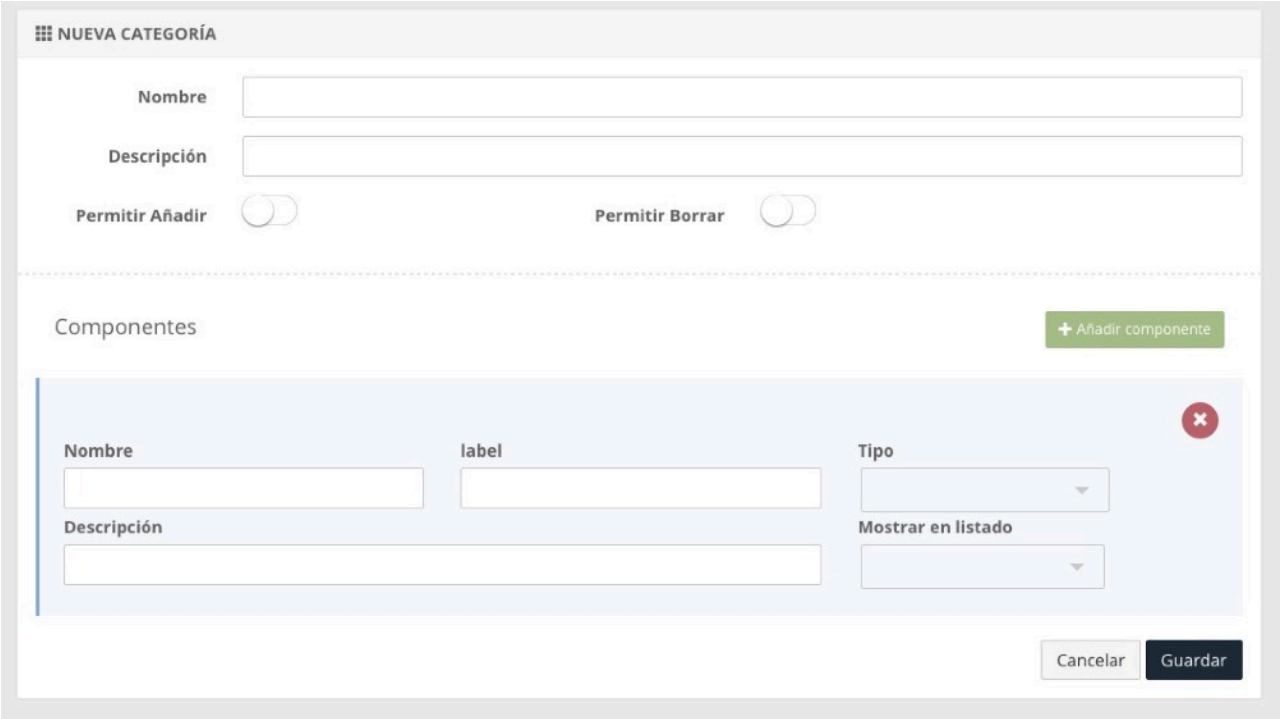


Gráfico 24: formulario categorías

En la zona superior se encuentran los campos comunes para todas las categorías y en la zona inferior nos permite gestionar y añadir los componentes de la categoría. Para cada componente habrá que indicar el tipo de campo de texto. Nombre y descripción son para uso interno mientras que label se utilizará tanto en los listados como en el cliente público o frontend. La opción mostrar en el listado indica si este campo debe aparecer en los listados o no.

Tanto si guardamos como si cancelamos el sistema nos llevará de nuevo al listado de categorías.

El formulario de edición de categorías es muy similar al de creación de categorías con la salvedad de que los datos de la categoría que queremos modificar serán precargados.

Al pulsar en el botón de borrar categoría se mostrará una ventana modal para confirmar que deseamos eliminar la categoría.

Gráfico 25: Confirmación eliminar

ADMINISTRAR CONTENIDO

La estructura de la sección para la administración de contenido es muy similar a la de administración de categorías. En una primera pantalla se muestra un listado del contenido disponible para la categoría seleccionada indicando el nombre y las acciones disponibles. En función de cómo haya sido la categoría se mostrará los botones de editar y borrar o sólo el de editar. El botón para añadir contenido también dependerá de la configuración de la categoría.

Gráfico 26: lista de contenido.

Al pulsar en el botón de borrar contenido se mostrará una ventana modal para confirmar que deseamos eliminar la contenido.

El formulario para la creación de contenido inicialmente nos muestra un selector de categorías. Esto es así porque el contenido del formulario variará en función de la categoría seleccionada.

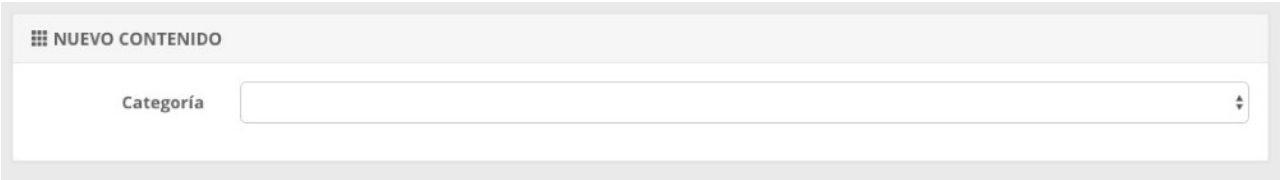


Gráfico 27: selector de categorías.

Una vez seleccionada la categoría se muestra un formulario para rellenar todos los tipos de contenido que componen la categoría.

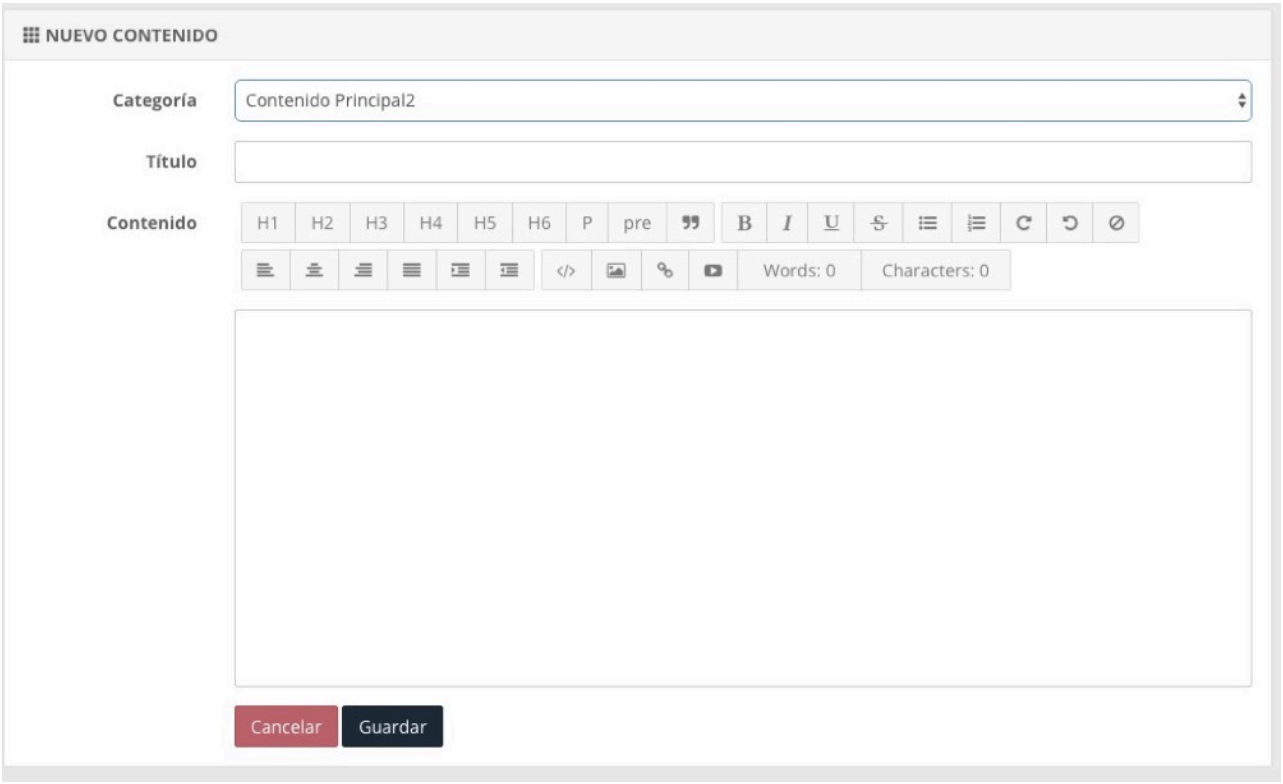


Gráfico 28: formulario de contenido.

El formulario de edición de contenido es muy similar al de creación con la salvedad de que los datos del contenido que queremos modificar serán precargados.

ADMINISTRACIÓN DE USUARIOS

De nuevo la estructura de la sección para la administración de usuarios es muy similar a la de administración de categorías. En una primera pantalla se muestra un listado de los usuarios con las opciones para editar, borrar o añadir nuevo.

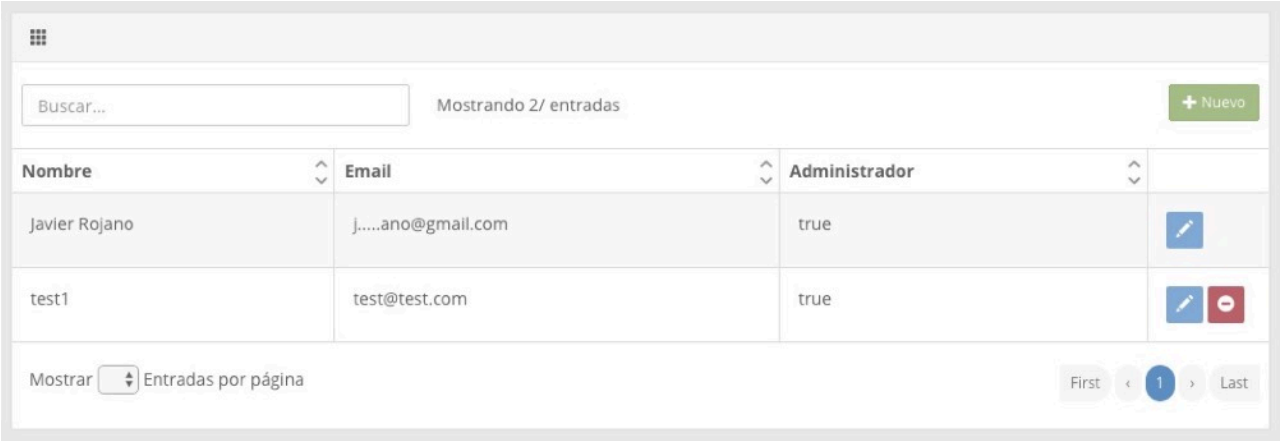


Gráfico 29: Lista de usuarios.

Las pantallas para la creación y la edición de usuarios muestran un formulario para introducir los datos del usuario.

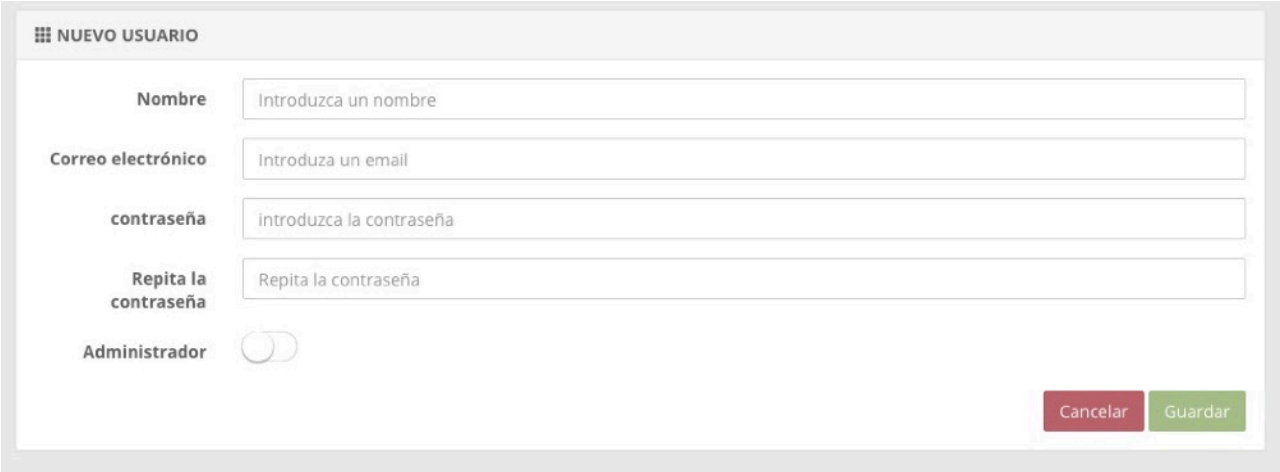


Gráfico 30: formulario de usuarios.

Al pulsar en el botón de borrar contenido se mostrará una ventana modal para confirmar que deseamos eliminar la contenido.

4.4 Implementación

Una vez finalizado el diseñado en detalle todo el sistema iniciaremos la implementación del mismo con las tecnologías elegidas. Dicha implementación se divide en la implementación del servidor y la implementación de la interfaz de administración.

Para la implementación se ha usado la pila de aplicaciones **MEAN (MongoDB - Express - AngularJS - Node.JS)**

5.4.1 Implementación del Servidor

Dentro de la pila de aplicaciones MEAN las tecnologías que se aplican al servidor son MongoDB, Express y Node Js. La implementación se ha llevado a cabo siguiendo el diseño de los recursos de la API expuestos en la sección 5.3.1, desarrollando la funcionalidad de cada recurso en cada una de las capas.

El siguiente gráfico muestra la arquitectura y tecnologías empleadas para el desarrollo del servidor.

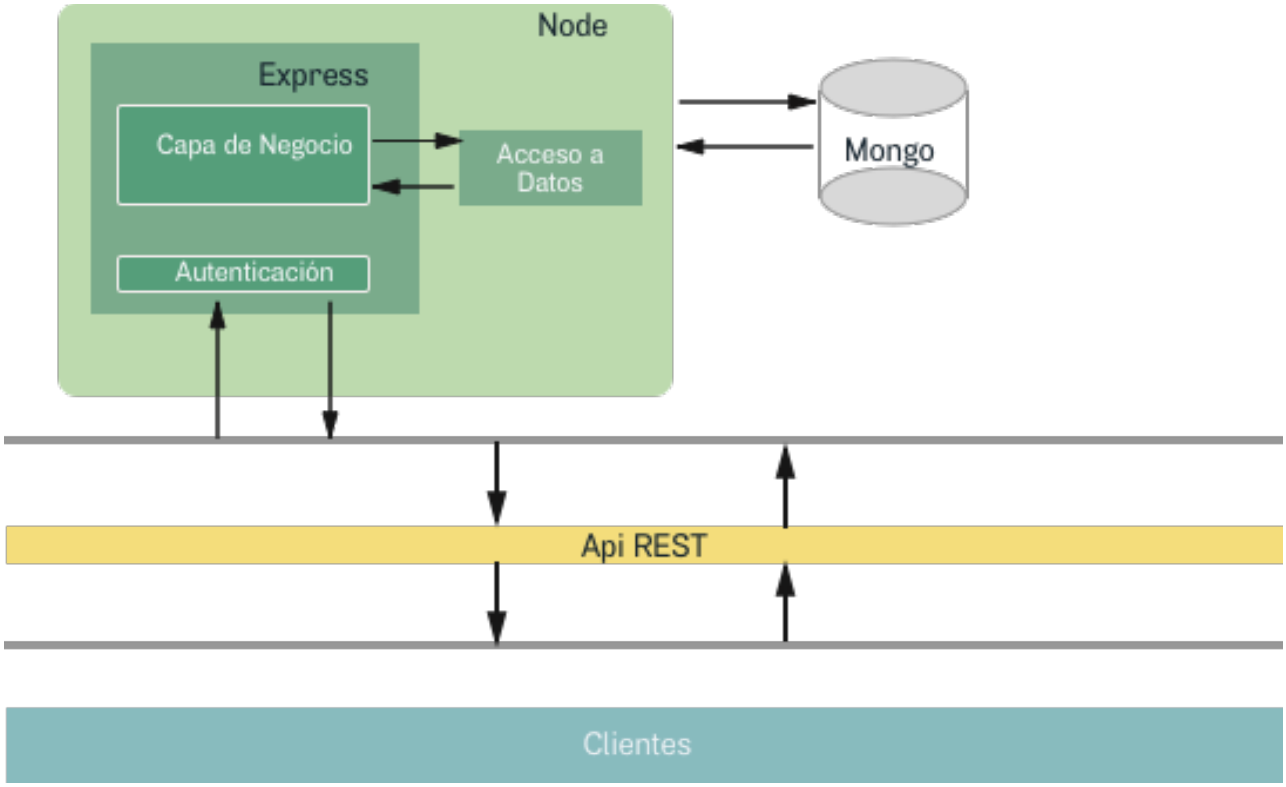


Gráfico 31: Arquitectura y tecnologías del servidor

Como se puede apreciar en el anterior diagrama este tipo de arquitectura es muy flexible y puede ser utilizada por cualquier cliente que consuma datos de una API RESt.

Esta es la estructura de directorios de nuestro servidor :

Gráfico 32: Estructura de archivos del servidor

En el directorio raíz nos en con los archivos packages.json, app.js y config.js.

Package.json es un archivo generado por el gestor de paquetes de npm donde queda reflejada la configuración del proyecto node (nombre, autor, versión, dependencias, etc).

En el archivo **config.js** almacenaremos las variables de configuración de nuestra aplicación.

```
module.exports = {
 'secret': ‘PFCsupersecureSecret',
 'database': 'mongodb://localhost:27017/pfc_cms',
  'languages': ['es','en']
};
```

El punto de acceso a la aplicación del servidor es el archivo **www** dentro del directorio bin. En este archivo se enlaza con el archivo principal de la aplicación **app.js** y se inicia el servidor.

El archivo **app.js** es el archivo principal de la aplicación a continuación se exponen y explican las líneas de código más importantes.

```
…

// conectamos con la base de datos
var mongoose = require('mongoose');
mongoose.connect(config.database);

// Inicializamos express lo asignamos a la variable app y habilitamos CORS para todas las //
   solicitudes
var app = express();

app.all('*', function(req, res, next) {
   res.header('Access-Control-Allow-Origin', '*');
   res.header('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE,OPTIONS');
res.header('Access-Control-Allow-Headers','Content-type,Accept,X-Access-Token,X-Key');
   next();
});
…

//Enlazamos con la lista de rutas de nuestra aplicación
app.use('/', require('./routes'));

…

// exportamos el objeto app para que pueda ser accedido desde otros módulos
module.exports = app;
```

Cualquier cliente que quiera acceder a los datos debe lanzar primero una petición REST a la ruta /login con un usuario y contraseña válidos. El servidor validará las credenciales y en caso afirmativo devolverá al cliente un objeto usuario junto con token de acceso.

El token tendrá asociado una fecha de caducidad. El cliente debe almacenarlo localmente y enviarlo con cada nueva petición que realice.

Tan pronto como se recibe una petición de cliente a cualquiera de las rutas definidas en la API REST, esta es redirigida al middleware de autenticación. En express las funciones de middleware son funciones que tienen acceso al objeto de solicitud (req), al objeto de respuesta (res) y a la siguiente función de middleware en el ciclo de solicitud/respuestas de la aplicación. La siguiente función de middleware se denota normalmente con una variable denominada next.

El middleware de autenticación es el responsable de autenticar al cliente para ello deberá procesar el token, si este no estuviera presente o no es válido responderá con un error HTTP 401. Si el token es válido y no ha expirado se invocará a la siguiente función de middleware.

El módulo de autenticación está compuesto por el archivo **/routes/auth.js** y el middleware de express definido en **/helpers/authentication.js**

En **auth.js** lo primero que hacemos es obtener todas las dependencias para la autenticación: la lista de modelos de moongoose, la configuración de la aplicación y jwt-simple. jwt-simple es un modulo para generar tokens web json

```
var model=require('../model');
var logger = require('../logger');
var mongoose = require('mongoose');
var config = require('../config');
var jwt    = require('jwt-simple');
```

A continuación definimos el objeto auth que tendrá los siguientes métodos

```
var auth={
        authenticate: function(){...},
        validate: function(){...},
        validateUser: function(){...},
        genToken: function(){...},
        expiresIn: function(){...}
}
```

La función **authenticate** obtiene las credenciales del usuario del cuerpo de la petición y las válida. Si las credenciales son válidas se genera y se devuelve un token.

La función **validate** tiene como parámetros de entrada un email, una contraseña y una función de callback. Busca en la base de datos el usuario identificado por el email que obtiene como primer parámetro y verifica si la contraseña, pasada como segundo parámetro, coincide con la almacenada en la base de datos. Una vez termina la búsqueda del usuario hace una llamada a la función callback indicando si la validación ha sido correcta o no y caso afirmativo devuelve los datos del usuario

validateUser busca en la base de datos si el email pasado por parámetro existe. El segundo parámetro es la función de callback que se invocará cuando termine la búsqueda.

gentoken utiliza jwt encode para generar y devolver un token con la información de usuario pasada como parámetro.

```
var expires = auth.expiresIn(1);
var token = jwt.encode(
              {
          exp: expires,
          user: user
        },
       config.secret);
```

expiresIn obtiene como parámetro de entrada un número de días y devuelve los milisegundos desde la fecha actual hasta el número de días indicado.

Como ya hemos comentado el módulo de autenticación hace uso de un middleware definido en /**helpers/authentication.js**

```
function ensureAuthorized(req, res, next) {
        var token = req.body.token || req.query.token || req.headers['x-access-token'];

        if(token){
                try{
                        var decoded = jwt.decode(token, app.get('superSecret') );

                        //Comprobamos que el token no haya expirado
                        if (decoded.exp <= Date.now()) {
                    res.status(401);
                    res.json({
                      "status": 401,
                      "message": "Token Expired"
                    });
                    return;
                   }

                    //si no ha expirado
                    validateUser(decoded.user.email, function(user){
                        // if everything is good, save to request for use in other routes
                                if(user){
                                    req.decoded = decoded;
                                    next();
                                }else{
                                    res.status(401);
                                            res.json({
                                              "success": false,
                                               "message": "Usuario no válido"
                                             });
                                }

                            });


                }catch (err) {
                res.status(500);
                res.json({
                  "success": false,
                  "message": "Oops something went wrong",
```

```
        "error": err
      });
    }
  }else{
        res.status(401);
    res.json({
      "success": false,
      "message": "Invalid Token or Key"
    });
    return;
  }
}
```

IMPLEMENTACIÓN DE LA API REST

Todas las rutas de la api se definen el archivo **/routes.index.js** En este archivo se definen todas las rutas de nuestra API REST, se enlazan con su controlador correspondiente y se verifica se indica si son accesibles públicamente o si por el contrario necesita autenticación.

Como ya hemos comentado anteriormente nuestra API REST se basa en express por tanto lo primero que debemos hacer es importar express y express.Router. Como alguna de las rutas serán públicas y otras privada también tendremos que importar el middleware para la autenticación.

```
var express = require('express');
var router = express.Router();
var helpers = require('../helpers/authentication');
```

Cada recurso tiene su propio controlador definido en archivos independientes dentro de la carpeta routes por tanto debemos importarlos para poder asociarlo a su ruta correspondiente.

```
var auth = require('./auth');
var user = require('./users');
var categories = require('./categories');
var fieldTypes = require('./fieldTypes');
var contentItems = require('./contentItems');
var shared = require('./shared');
```

A continuación se definen todas las rutas de la aplicación. La definición de las rutas se sigue el siguiente esquema:

```
router.METHOD(PATH,helpers.ensureAuthorized,CONTROLADOR)
```

Donde:

- router es una instancia de express.Router().
- METHOD es el tipo de petición HTTP.
- PATH es la ruta de acceso
- helpers.ensureAuthorized es el middleware de autenticación. Es opcional de forma de que si está presente antes de pasar el control al controlador verifica que la petición va acompañada de un token válido.
- CONTROLADOR es la funcinón que se va a hacer cargo de manejar la ruta.

```
/*
 * auth Routes
 */
router.post('/api/authenticate', auth.authenticate);

/*
 *        User Routes
 */
router.get('/api/users',helpers.ensureAuthorized,user.getAll);
router.get('/api/users/:id',helpers.ensureAuthorized,user.getOne);
router.post('/api/users',helpers.ensureAuthorized,user.create);
router.put('/api/users/:id',helpers.ensureAuthorized,user.update);
router.delete('/api/users/:id',user.delete);

/*
 *        fieldTypes Routes
 */
router.post('/api/fieldtypes',helpers.ensureAuthorized, fieldTypes.create);
router.get('/api/fieldtypes/:category_id?',fieldTypes.getAll);
```

```
/*
 *        Categories Routes
 */
router.post('/api/categories',helpers.ensureAuthorized, categories.create);
router.get('/api/categories',categories.getAll);
router.get('/api/categories/:category_id',categories.getOne);
router.get('/api/categories/:id/content_items',contentItems.getAll);
router.put('/api/categories/:category_id',helpers.ensureAuthorized,categories.update);
router.delete('/api/categories/:category_id',helpers.ensureAuthorized,categories.delete);

/*
 *        Content Items Routes
 */
router.get('/api/content_items/stats',contentItems.stats);
router.get('/api/content_items',contentItems.getAll);
router.get('/api/content_items/:id',contentItems.getOne);
router.post('/api/content_items',helpers.ensureAuthorized,contentItems.create);
router.put('/api/content_items/:id',helpers.ensureAuthorized,contentItems.update);
router.delete('/api/content_items/:id',helpers.ensureAuthorized,contentItems.delete);
```

IMPLEMENTACIÓN DE LA CAPA DE NEGOCIO

Cada ruta tiene asociado su propio controlador. Los controladores están ubicados en la carpeta routes y agrupados por recursos de forma que por cada recurso se ha creado un archivo que contiene todos sus controladores.

Gráfico 33: controladores del servidor

Como el archivo auth.js ya se explicó en la sección anterior comenzaremos por el archivo **categories.js**.

Categoríes.js

Como su nombre indica, este archivo contendrá toda la lógica relacionada con el manejo de categorías. Todos los controladores comparten una estructura similar.

En la zona superior se importan todas las dependencias.

```
var model=require('../model');
var logger = require('../logger');
var mongoose = require('mongoose');
```

A continuación se define un objeto que contendrá todos los controladores del recurso.

```
var categories={
    getAll: function(req,res){
        ...
    },
    getOne: function(req,res){
        ...
    },
    create: function(req, res){
        ...
    },
    update: function(req,res){
        ...
    },
    delete: function(req,res){
        ...
    }
}
```

Por último se exporta el objeto categoríes para que se accesible desde otras partes de la aplicación.

```
module.exports = categories;
```

Vamos a pasar a describir cada uno de los controladores definidos en el objeto categories.

categories.getAll. Se trata del controlador para la ruta /**api/categories/**. Esta ruta de la Api devuelve una lista de las categorías definidas en el sistema. Para ello el controlador se apoya en el modelo category. Tal y como se explicará más adelante la capa de acceso a datos se ha implementado haciendo uso de mongoose, que nos permite modelar los datos de la aplicación.

```
model["category"]
        .find()
        .populate("components.type_id")
        .execute(...);
```

En el código que se muestra arriba, lo primero que hacemos es obtener el modelo para las categorías y le aplicamos el método find. Al no indicar ningún filtro, este método nos devolverá todas las categorías definidas en el sistema. Como nos interesa tener los detalles de los distintos componentes, que se encuentran en otra colección debemos popular el contenido referenciado por components.type_id. Por último indicamos que hacer cuando la consulta haya terminado. Si se produce un error devolveremos el error y si todo ha ido bien devolveremos los datos obtenidos.

```
function(err,items){
        if(err){
            res.status(500);
            res.json({
                success: false,
                message: err
            });
        }else{
            res.json(items);
        }
});
```

categories.getOne. se trata del controlador para la ruta **/api/categories/:id** . En primera instancia se obtiene el modelo para las categorías y se hace una búsqueda usando el método findOne con el id indicado en la ruta.

```
var id=mongoose.Types.ObjectId(req.params.category_id);
var category =  model["category"];
category
    .findOne({_id: id })
    .exec(findCategoryResult);
```

findCategoryResult devolverá un error 404 si no se encuentra la categoría o los datos de la categoría.

categories.create. Se trata del controlador para la ruta de creación de categorías (POST / api/categories).

```
var document=req.body;
var category = new model["category"](document);
  category.save(function(err, result){
     ...
  });
```

Las llamadas a esta ruta deben incluir en el body un objeto categoría, lo primero que hacemos es obtener el objeto y crear un nuevo documento categoría. A continuación usamos el método save para guardar los datos.

categories.uptade. Se trata del controlador para la ruta de modificación de categorías PUT **/api/categories/{id}**.

```
var data=req.body;
var id=mongoose.Types.ObjectId(req.params.category_id);
var category =  model["category"];
category.findOneAndUpdate({_id: id },data,function(err,doc){});
```

Las llamadas a esta ruta deben incluir en la ruta el id de la categoría a modificar y en el body los nuevos datos de la categoría. Lo primero que hacemos es obtener el objeto del body y el id de los parámetros de la petición. Los identificadores mongo son de tipo ObjectId y el id que nos llega por parámetro es una string por tanto debemos realizar la conversión para poder trabajar con el. A continuación usamos el método findOneAndUpdate para guardar los datos. FindOneAndUpdate tiene 3 parámetros, el primero es un json con el filtro para la búsqueda, el segundo son los datos que queremos actualizar y el tercero la función de callback.

categories.delete. Se trata del controlador de la ruta para eliminar categorías (**DELETE / api/categories/{id}**).

Antes de eliminar una categoría primero deberemos eliminar todo el contenido asociado a esa categoría. Para ello obtenemos el id de la categoría de la lista de parámetros y

haciendo uso del modelo contentItem eliminamos todos los documentos con ese category_id. Hasta que no termine este proceso no podemos proceder a eliminar la categoría por lo que en la función de callback llamamos a la función removecategory con el id de la categoría a eliminar.

```
function removecategory(id){
        model["category"].remove({_id:id}, function (err, result) {
        if (err) {
           res.status(500);
              res.json({
                 success: false,
                 message: err
              });
              return;
        } else {
           res.json(result);
        }
      });
      }
```

ContentItem.js

contentItems.getAll. Este controlador devuelve una lista de elementos de contenido y es usado por la rutas GET /api/categories/:id/contentItems y GET /api/contentItems. En el primer caso devuelve todo el contenido de una categoría determinada y en el segundo devuelve todo el contenido almacenado en el sistema.

```
var filter={};

 //comprobamos si se debe filtrar por categoría
 if(typeof req.params.id!== "undefined"){
     var id=mongoose.Types.ObjectId(req.params.id);
     filter={category_id: id};
  }

     getContentList(res,filter);
```

Primero definimos una variable filter que definimos como un objeto json vación. A continuación comprobamos si la ruta incluye un id de categoría y en caso afirmativo actualizamos el filtro. Por último se hace una llamada a la función getContentList pasando como parámetros el filtro y el objeto para manejar la respuesta. La función getContentList

obtiene una lista de los documentos que cumplen los requisitos del filtro y devuelve la lista de documentos populando los datos de la categoría y los componentes.

contentItems.getOne. Se trata del controlador para la ruta GET /api/categories/:id. Primero se obtiene el modelo para las contentItems y se hace una búsqueda usando el método findOne con el id indicado en la ruta.

```
contentItem.findOne({_id: id },function(err,doc){
  if(err) {
      res.status(500);
      res.json({
          success: false,
          message: err
      });
  } else {
      if(doc){
          res.json(doc.toJSON({ virtuals: true }));
      }else{
          res.status(404);
          res.json({
              success: false,
              message: "No se ha encontrado ningún artículo"
          });
      }
  }
});
```

Como particularidad comentar que en este modelo se han definido una serie de atributos virtuales para manejar los idiomas. En la siguiente sección entraremos en mas detalle sobre esta propiedad de mogoose. En este punto lo que nos interesa es que antes de procesar la respuesta debemos habilitar los virtuales.

```
res.json(doc.toJSON({ virtuals: true }));
```

contentItems.create. se trata del controlador para la ruta de creación de contenido (POST /api/contentItems).

Todo contenido debe pertenecer a una categoría para ello primero validamos la petición.

Antes de guardar el nuevo documento debemos comprobar si este incluye algún campo de tipo linkFile d para procesar el upload. Para ello se define la función checkFile.

```
checkFile(document,req,res,function(res){
        var contentItem = new model["contentItem"](document);
        contentItem.save(function(err, result){
                ...
})
```

```
var checkFile=function(document,req,res,callback){

   for(var i=0; i<document.languages.length;i++){
      var language=document.languages[i];
      for(var j=0; j<language.components.length;j++){
         var component=language.components[j];

         if(typeof component.urlFile=="object"){
            var currenLang=i;
            var currentComponent=j;
            var base64File=component.urlFile.base64;
            var decodedFile = new Buffer(base64File, 'base64');
            try{
               fs.writeFileSync('public/uploads/'+component.urlFile.filename, decodedFile);
            }
            catch(error){
               res.send(error,500);
            }

            var url=req.protocol + "://" + req.get('host')+'/
uploads/'+document.languages[currenLang].components[currentComponent].urlFile.filename;
            document.languages[currenLang].components[currentComponent].urlFile=url;
         }
      }
   }
   callback(res);
}
```

Esta función recorre el documento pasado por parámetro en busca de si alguno de los componentes de cualquiera de los idiomas es de tipo urlFile. En caso afirmativo lo procesa y lo almacena en disco.

contentItems.update. se trata del controlador para la ruta de creación de contenido (PUT /api/contentItems/:id).

```
var data=req.body;
...
var id=mongoose.Types.ObjectId(req.params.id);
 var contentItem =  model["contentItem"];
  contentItem.findOne({_id: id }).exec(function(err,doc){
      if(err){
                        ...
            else{
                        checkFile(data,req,res,function(res){
                                ...
                        })
            }

});
```

Nuevamente se hace uso de la función checkFile para manejar los uploads de archivos

contentItems.delete. Se trata del controlador de la ruta para eliminar contenido (DELETE /api/contentItem/{id}).

fieldtypes.js

fieldTypes.getAll. Se trata del controlador para ruta para obtener los tipos de componentes de contenido definidos en el sistema. GET /api/fieldtypes.

```
model["fieldType"].find().exec(function(err,items){
        res.json(items);
    });
```

Users.js

User.getAll. controlador para la ruta de obtención de la lista de usuarios. (GET /api/users). Hace uso del método find para obtener la lista completa de usuarios.

User.getOne. controlador para la ruta de obtención de la lista de usuarios. (GET /api/users/id). Hace uso del método findOne para obtener el documento de usuario que se indica en la lista de parámetros.

User.create. controlador para la ruta de creación de usuarios (POST /api/users). Lo primero que hace es verificar que no hay otro usuario con el mismo email en la base de datos. Si no existe el usuario los crea y genera un token.

```
var  User=model["user"];
User.findOne(
{"email": req.body.email},
    function(err,user){
            if(err){...}
            //si el usuario ya existe
            if(user){
                    res.status(500);
                    res.json({
                            success: false,
                            message: "El usuario ya existe"
                    });
                    return;
            }else{
                    var userModel=new model["user"]();
                    userModel.email=req.body.email;
            userModel.name=req.body.name;
                    userModel.password=userModel.generateHash(req.body.password);
                    userModel.save(function(err,user){
res.json(genToken(user));
});

            }

});
```

User.update. controlador para la ruta de modificación de usuarios. (PUT /api/users/:id). Hace uso de los métodos findbyId y save para obtener y modificar el usuario.

User.delete. controlador para la ruta de elimianación de usuarios. (DELETE /api/users/:id). Hace uso del método remove para eliminar el usuario.

IMPLEMENTACIÓN DE LA CAPA DE ACCESO A DATOS

La capa de acceso a datos se ha desarrollado haciendo uso de Mongoose. Mongoose es una herramienta de modelado de objetos mongoDB para nodeJS que usa una solución basada en la definición de esquemas para modelar los datos de la aplicación. Incluye validación, verificación de tipos de datos, herramientas para realizar consultas etc.

En Mongoose todo se basa en la definición de esquemas. Todo esquema está relacionado con una colección MongoDB y define la estructura que tendrán los documentos que contiene.

Los esquemas o modelos de datos de nuestra aplicación están definidos en la carpeta /model. En ella hemos creado un archivo por cada recurso de nuestra API.

Gráfico 34: estructura de ficheros modelo de datos

FIELDTYPE

El modelo fieldType representa los tipos de componentes de contenido. Como ya hemos explicado todo contenido tiene que estar asociado a una categoría y todos los elementos de una misma categoría tienen los mismo tipos componentes de contenido.

```
var Resource = new Schema({
  name                : { type: String, required: true},
```

```
    desc                : { type: String },
    opts                : { type: Array, default: []}
});
```

Los fieldTypes tienen un nombre una descripción y una serie de opciones. A continuación se presenta un ejemplo de fieldType.

```
{
    "_id" : ObjectId("54da55b178dc0e5fc69dd1c8"),
    "name" : "cms-linkfile",
    "desc" : "enlace a una url o fileUpload",
    "opts" : [
        {
            "name" : "mime_types",
            "desc" : "Mime Types",
            "label" : "Accepted Mime Types",
            "options" : [
                {
                    "text" : "application/pdf"
                },
                ,
                {
                    "text" : "application/msword"
                },

            ],
            "type" : "TAGS INPUT"
     }
```

Los documentos de la colección Categories siguen el siguiente esquema.

```
var Resource = new Schema({
    name                : { type: String, required: true},
    desc                : { type: String },
    canDelete : { type: Boolean, default: true},
    canAdd              : { type: Boolean, default: true},
    items       : { type: Number, default: 0},
    components : [{
        name : { type: String, required: true},
        desc : { type: String },
        label: { type: String },
        show : { type: Boolean, default: true},
        extra: {type: Schema.Types.Mixed},
```

```
        type_id: {type:Schema.Types.ObjectId, ref: 'field_types' }
    }]
});
```

Las categorías tienen una serie de campos comunes (nombre, desc, canDelete, canAdd e items) y una serie de componentes que pueden variar de una categoría a otra. Es el usuario el que define los componentes de una categoría en el momento de su creación.

En mongoDB las relaciones entre colecciones se realiza por referencia. En este caso los componentes de una categoría se relacionan con la colección fieldTypes a través del campo type_id.

```
type_id: {type:Schema.Types.ObjectId, ref: 'field_types' }
```

Cuando se establece una referencia en mongoose al hacer queries sobre una colección se pueden popular los datos de la colección referenciada.

CONTENTITEMS

Los documentos de la colección Categories se han definido haciendo uso del siguiente esquema.

```
var Resource = new Schema({
    category_id: {type:Schema.Types.ObjectId, ref: 'categories' },
    created: { type: Date, default: Date.now },
    updated: { type: Date, default: Date.now },
    languages:[{language:String, components:[]}]
});
```

Como se puede observar todo elemento de contenido está referenciado a una categoría. Como el sistema es multi idioma cada elemento de contenido contiene un array con los diferentes idiomas. Cada idioma a su vez contiene un array con los diferentes componentes de la categoría. En este esquema no se hace referencia a ningún tipo de restricción sobre los componentes del contenido, es responsabilidad del cliente construir el objeto con los componentes correspondientes a la categoría referenciada en category_id.

El modelo de datos de categoría tiene un campo items que indica el número de elementos de contenido que contiene. Para mantener actualizado ese contador vamos a definir en el modelo de datos de contentItems una serie de funciones que actualizan el contador cuando se crean y borran elementos de contenido.

```
Resource.post('save', function (doc){
        var update={ $inc: { items: 1 }};
        var callback=function (err, result){
                ...
          }
              Category.update({ _id: doc.category_id }, update, callback);

});

Resource.post('remove', function (doc,next){
        var update={ $inc: { items: -1 }};
     var callback=function (err, result){
          ...
        }
        next();
     }
     Category.update({ _id: doc.category_id }, update, callback);
});
```

Para facilitar el manejo de los distintos idiomas por parte de los clientes se han definido un campo virtual por cada idioma, de forma que podamos acceder a contentItem.es para ibtener el contenido en español.

```
var esVirtual=Resource.virtual('es');
esVirtual.get(esVirtualGet);
esVirtual.set(esVirtualSet);

function esVirtualGet(){
        var data;
        this.languages.forEach(function(element){
                if(element.language=="es"){
                        data= element.components;
                }
        })
        return data;
}
```

```
    function esVirtualSet(data){
        var languages=this.languages;
        var found=false
        for(var i in languages){
            if(languages[i].language=="es"){
                this.languages[i].components=data.components;
                found=true;
            }
        }
        if(!found){
            this.languages.push(data);
        }
}
```

USERS

Los documentos de la colección Categories se han definido haciendo uso del siguiente esquema.

```
var Resource = new Schema({
    email: String,
    name: String,
    password: String,
    admin: {type:Boolean, default: false},
    token: String
});
```

A parte se han definido dos métodos nuevos en el esquema para la gestión de las contraseñas.

```
// método para verificar si una contraseña es válida
Resource.methods.validPassword = function(password) {
        return bcrypt.compareSync(password, this.password);
};
// Método para encriptar una contraseña
Resource.methods.generateHash = function(password) {
    return bcrypt.hashSync(password, bcrypt.genSaltSync(8), null);
};
```

4.4.2 Implementación del Panel de Administración

El panel de control o interfaz gráfica hace uso de Slim. Slim es una plantilla de panel de control basada en Angular JS y bootstrap.

La estructura básica de la plantilla es:

```
Slim/
├── Slim/                        ──> Contains the develop version of Slim Admin Template.
│   ├── client/                              └──> All front end files are in this folder.
│   │   ├── fonts/                               └──> contains icon fonts.
│   │   ├── i18n/                                └──> files for i18n (Internationalization)
│   │   ├── images/
│   │   ├── scripts/                             └──> All the custom scripts
│   │   ├── styles/                              └──> All the custom stylesheets (Scss files)
│   │   ├── views/                               └──> All the custom templates
│   │   ├── favicon.ico
│   │   └── index.html
│   ├── .bowerrc
│   ├── .editorconfig
│   ├── .gitattributes
│   ├── .jshintrc
│   ├── .travis.yml
│   ├── bower.json
│   ├── Gruntfile.coffee
│   └── package.json
├── dist/                         ──> The distribution (production)  version of Slim Admin
└── documentation/                ──> Where the help documentation is.
```

Gráfico 35: estructura de ficheros del panel de control

Vamos a describir alguno de los archivos y carpetas principales:

El código fuente de la plantilla se encuentra en .Slim/Client

En la raíz de esta carpeta nos encontramos los principales archivos de configuración de la aplicación:

bower.json Este archivo te sirve para especificar de una manera formal todas las dependencias que tiene tu proyecto. De esa manera podremos usar Bower para que instale o actualice todas las dependencias de una sola vez. Este archivo tienen una sintaxis JSON, indicando una serie de campos que necesita para definir tu proyecto y sus dependencias.

Gruntfile.coffee Se trata del fichero de configuración del automatización de tareas de grunt. En este archivo se especifican y configuran las tareas que se desean automatizar y los plugins grunt que se van a utilizar.

Package.json Es el archivo que emplea npm para almacenar los metadatos del proyecto. En el se listan las dependencias de nuestro proyecto incluido grunt y sus módulos.

Bower. Bower es un gestor de dependencias para el desarrollo web frontend que te facilitará la tarea de instalar y mantener al día librerías y frameworks en tus proyectos.

Slim es un panel de control multipropósito por lo que el archivo de dependencias original es muy completo, incluyendo numerosos módulos que nosotros no vamos a usar en nuestro proyecto.

Una vez optimizada para las necesidades de nuestro proyecto la lista de dependencias es:

```
{
  "name": "slim",
  "version": "3.0.0",
  "dependencies": {
    "jquery": "~2.1.0",
    "angular": "~1.5.0",
    "angular-route": "~1.4.0",
    "angular-animate": "~1.4.0",
    "angular-aria": "~1.4.0",
    "angular-bootstrap": "~0.14.0",
    "font-awesome": "~4.5.0",
    "jquery.slimscroll": "~1.3.3",
    "angular-scroll": "~1.0.0",
    "angular-translate": "~2.8.0",
    "angular-translate-loader-static-files": "~2.8.0",
    "angular-loading-bar": "~0.8.0",
    "toastr": "~2.1.0",
    "seiyria-bootstrap-slider": "~5.2.0",
    "jquery-steps": "~1.1.0",
    "textAngular": "~1.5.0",
    "angular-ui-tree": "~2.11.0",
    "ng-tags-input": "~3.0.0",
    "bootstrap-file-input": "~1.0.0",
    "echarts": "~2.2.7",
```

```
    "ngECharts": "ngecharts#~0.1.1",
    "angular-validation-match": "~1.7.1",
    "angular-ui-router": "ui-router#~0.2.18",
    "angular-resource": "~1.5.0"
  },
  "devDependencies": {
    "bourbon": "~4.2.0",
    "bootstrap-sass-official": "~3.3.0"
  },
  "resolutions": {
    "angular": "~1.4.0"
  }
}
```

AUTOMATIZACIÓN CON GRUNT

Grunt.js es una librería JavaScript que nos permite configurar tareas automáticas. Slim incluye los archivos de configuración de grunt: Gruntfile.cofee y package.json.

La lista de tareas disponible a través de grunt es:

grunt serve ejecuta un servidor web local con browser sync a partir de nuestros archivos originales. De esta manera no tendremos que actualizar el navegador cada vez que hacemos un cambio en el código.
grunt build Genera una versión optimizada de la aplicación en la carpeta dist.
grunt serve:dist Ejecuta un servidor web local con la versión optimizada de la aplicación.

CONFIGURACIÓN DEL ENTORNO DE DESARROLLO

Para poder trabajar con slim primero deberemos tener instalado en nuestro entorno de desarrollo:

- Node.js
- Yeoman,Bower, Grunt-cli.
- Ruby, Sass , Compass.

Una vez tengamos todas las herramientas necesarias instaladas debemos descargar e instalar todas las dependencias de nuestro proyecto, para ello haremos uso de npm y bower.

```
# npm install
# bower install
```

La estructura de carpetas inicial dentro de la carpeta clients es la siguiente

```
client/
├── fonts/      → contiente las fuentes para glyphicons
├── i18n/       → Contiene los archivos json para la traducción
├── images/     → donde se almacenan las imágenes
├── scripts/    → donde debemos incluir nuestros scripts
├── styles/     → contiene las hojas de estilo
├── views/      → contiene los archivos de las plantillas (html)
├── favicon.ico
└── index.html → archivo html principal
```

Esta estructura de archivos no está mal como punto de partida pero he preferido realizar una serie de modificaciones para adaptarlo a las recomendaciones de la guía de estilo para Angular 1 (https://github.com/johnpapa/angular-styleguide/tree/master/a1).

Los principales puntos que se ha tenido en cuenta son:

- Definir un solo componente por fichero.
- Encapsular los componentes Angular en una función invocada a continuación (IIFE de las siglas en inglés - Immediately Invoked Function Expression).

En vez de agrupar los archivos en carpetas según su tipología, lo que se ha hecho es agruparlos según su funcionalidad. Por ejemplo todos los archivos relacionados con la autorización están agrupados dentro de la misma carpeta.

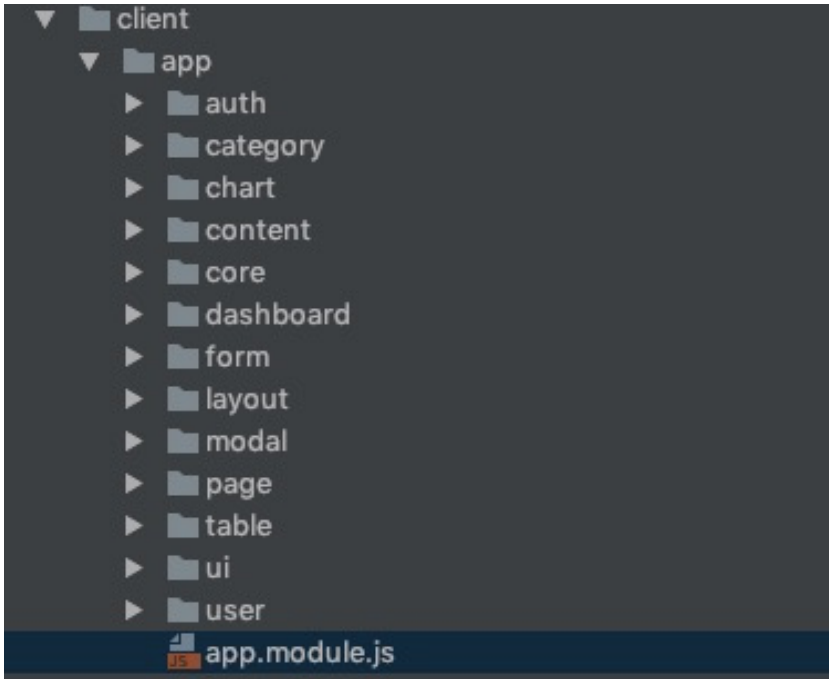


Gráfico 36: estructura de ficheros del panel de administración 1

Como se puede observar para la autorización hemos implementado 3 archivos javascript (un controlador, una factoría y un módulo) y una plantilla html.

La estructura de directorios resultante es la siguiente:

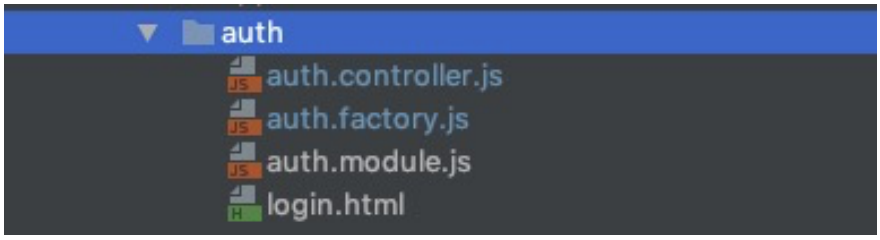


Gráfico 37: estructura de ficheros del panel de administración 2

La estructura básica de la plantilla (index.html) es mu sencilla. Se compone de tres secciones: La Cabecera en donde se enlazan todas las hojas de estilos. El cuerpo, que contendrá el contenido de nuestra aplicación y el pie que cargará todos los scripts.

La Cabecera

```
<!doctype html>
<!--[if lt IE 8]>      <html class="no-js lt-ie8"> <![endif]-->
```

```
<!--[if gt IE 8]><!--> <html class="no-js"> <!--<![endif]-->
   <head>
      <meta charset="utf-8">
      <meta http-equiv='X-UA-Compatible' content='IE=edge'>
      <title>Web Application</title>
      <meta name="description" content="Responsive Admin Web App">
      <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-
scale=1.0">

      <!-- <link href='http://fonts.googleapis.com/css?family=Open+Sans:300italic,400italic,600italic,
400,600,300,700' rel='stylesheet' type='text/css'> -->
      <!-- Needs images, font... therefore can not be part of main.css -->
      <link rel="stylesheet" href="fonts/themify-icons/themify-icons.min.css">
      <link rel="stylesheet" href="bower_components/font-awesome/css/font-awesome.min.css">
      <!-- end Needs images -->


         <!-- build:css({.tmp,client}) styles/main.css -->
         <link rel="stylesheet" href="styles/bootstrap.css">
         <link rel="stylesheet" href="styles/ui.css">
         <link rel="stylesheet" href="styles/main.css">
         <!-- endbuild -->

   </head>
```

El cuerpo

```
   <body data-ng-app="app"
         id="app"
         class="app"
         data-custom-page
         data-off-canvas-nav
         data-ng-controller="AppCtrl"
         data-ng-class=" {'layout-boxed': admin.layout === 'boxed'} "
         >
        <!--[if lt IE 9]>
           <div class="lt-ie9-bg">
              <p class="browsehappy">You are using an <strong>outdated</strong> browser.</p>
                    <p>Please <a href="http://browsehappy.com/">upgrade your browser</a> to
   improve your experience.</p>
           </div>
        <![endif]-->
```

```
<section data-ng-include=" 'app/layout/header.html' "
        id="header"
        class="header-container "
        data-ng-class="{ 'header-fixed': admin.fixedHeader,
                        'bg-white': ['11','12','13','14','15','16','21'].indexOf(admin.skin) >= 0,
                        'bg-dark': admin.skin === '31',
                        'bg-primary': ['22','32'].indexOf(admin.skin) >= 0,
                        'bg-success': ['23','33'].indexOf(admin.skin) >= 0,
                        'bg-info-alt': ['24','34'].indexOf(admin.skin) >= 0,
                        'bg-warning': ['25','35'].indexOf(admin.skin) >= 0,
                        'bg-danger': ['26','36'].indexOf(admin.skin) >= 0
        }"></section>

<div class="main-container">
    <aside data-ng-include=" 'app/layout/sidebar.html' "
          id="nav-container"
          class="nav-container"
          data-ng-class="{ 'nav-fixed': admin.fixedSidebar,
                          'nav-horizontal': admin.menu === 'horizontal',
                          'nav-vertical': admin.menu === 'vertical',
                          'bg-white': ['31','32','33','34','35','36'].indexOf(admin.skin) >= 0,
                          'bg-dark': ['31','32','33','34','35','36'].indexOf(admin.skin) < 0
          }">
    </aside>

    <div id="content" class="content-container">
        <section ui-view data-ng-view
                class="view-container {{admin.pageTransition.class}}"></section>
    </div>
</div>
```

En la etiqueta body observamos que se encuentra el atributo data-ng-app, que representa la directiva angular ngApp. Esta directiva sirve para etiquetar el elemento html que angular considerará como el elemento raíz de nuestra aplicación.

Con data-ng-controller="AppCtrl" indicamos cual es controlador principal de la aplicación.

Tanto el navbar como la barra lateral tienen sus propios arhivos html independientes, para incluirlos en el index html se hace uso de data-ng-include.

Silm usa el módulo ng-route para gestionar la navegación de la aplicación por las diferentes vistas. Este módulo tiene algunas carencias por lo que he decidido sustituirlo por ui-router. Este módulo se apoya en el concepto de "estado" para definir la navegación entre vistas. La etiqueta ui-view indica a ui-router donde se debe cargar el template una vez que un estado haya sido activado.

AUTENTICACIÓN

Como ya se ha comentado nuestra aplicación utiliza un esquema de autorización basado en tokens, por tanto el cliente debe obtener un token enviando las credenciales al servidor y almacenar dicho token para enviarlo en las siguientes peticiones al servidor. Para gestionar la autenticación nuestra aplicación tiene definido el módulo app.auth.

```
angular.module("app.auth")
  .config(config)
  .run(runFunction)
  .controller('loginCtrl',LoginCtrl);
```

Este módulo se compone de tres bloques. Un bloque de configuración donde definiremos los estados y rutas relativos al proceso de autenticación, un bloque de ejecución que gestiona la existencia o no del token y un controlador.

```
function config($stateProvider){
  $stateProvider
    .state('login',{
      url:'/login',
      templateUrl:'app/auth/login.html',
      controller: 'loginCtrl',
      controllerAs: 'vm'
    });
  return $stateProvider;
}
```

En el bloque de configuración se define un único estado para la pantalla de login que usará el template app/auth/login.html y el controlador loginCtrl.

```
function runFunction($log,$rootScope, $window, $location,$state, authenticationFactory){
        authenticationFactory.check();
        $rootScope.$on('$stateChangeStart',routeChangeStart);
```

El bloque de ejecución (run), al igual que el de configuración, se ejecuta antes de que se inicie la aplicación. Es el lugar perfecto para llevar a cabo las comprobaciones de autenticación. En el código se puede observar una llamada aauthenticationFactory.check que verifica si el usuario ya ha hecho login. Este servicio se define en auth.factory.js y se ha inyectado en el controlador.

A continuación se define una función que se ejecutará antes de cada cambio de estado, en la que se verifica si el usuario ya se ha identificado y en caso contrario lo redirige a la pantalla de login.

```
function routeChangeStart(event, toState, toParams, fromState, fromParams){
  var isLogged=authenticationFactory.isLogged;
  if(!isLogged && toState.name!="login"){
          $location.path('/login');
  }
}
```

El template del estado de login básicamente muestra un formulario para la introducción del email y la contraseña .

```
<input type="email" class="form-control input-lg" ng-model="vm.email">
<input type="password" class="form-control input-lg" ng-model="vm.password">
...
<a ng-click="vm.login()" class="btn btn-primary btn-lg btn-block text-center">Log in</a>
```

En angular el data-binding del controlador con las vistas se hace a través de el atributo ng-model. La directiva ng-clik nos permite personalizar el comportamiento del evento click, en este caso llamamos a la función vm.login del controlador. Esta función hace uso de userAuthFactory para enviar las credenciales al servidor y obtener un token.

```
var promise=userAuthFactory.login(vm.email,vm.password);
promise.success(loginSuccess);
```

Una vez obtenidos los datos los almacena localmente en el navegador del cliente y cambia a la vista de dashboard.

```
function loginSuccess(data){
  var user={
```

```
    name: data.user.name,
    email: data.user.email,
    admin: data.user.admin
  };
  authenticationFactory.isLogged = true;
  authenticationFactory.user = user;


  $window.sessionStorage.token = data.token;
  $window.sessionStorage.expires= data.expires;
  $window.sessionStorage.userName = user.name;
  $window.sessionStorage.userEmail = user.email;
  $window.sessionStorage.isAdmin = user.admin;
  $state.go('dashboard');
            ...
}
```

En el archivo auth.factory.js se definen una serie de servicios relacionados con la autenticación.

```
angular.module('app.auth')
        .factory('authenticationFactory', AuthenticationFactory)
        .factory('userAuthFactory', UserAuthFactory)
        .factory('tokenInterceptor', tokenInterceptor)
        .config(config);
```

Como ya se ha comentado el token se obtienen mediante userAuthFactory.login, pero este token lo tenemos que enviar en las cabeceras de cada nueva petición que se haga al servidor. Para ello hemos usado interceptores de AngularJS

```
function config($httpProvider){
        $httpProvider.interceptors.push('tokenInterceptor');
  }

function tokenInterceptor($q,$window,$injector){
        var interceptor={
                request: requestFunction,
                response: responseFunction,
                responseError: responseErrorFunction
        }
```

```
        return interceptor;

        function requestFunction(config){
                config.headers = config.headers || {};
                if ($window.sessionStorage.token) {
              config.headers['X-Access-Token'] = $window.sessionStorage.token;
              config.headers['Content-Type'] = "application/json";

        }
        return config || $q.when(config);
        }

        function responseFunction(response){
                return response || $q.when(response);
        }


  function responseErrorFunction(response){

    if(response.status === 401) {
        $injector.get('$state').transitionTo('login');
        return $q.reject(response);
      }
      else {
        return $q.reject(response);
      }
  }
}
```

En el código de arriba, cada petición es interceptada, y se agrega un encabezado y valor de autorización en los encabezados.

La carpeta core contiene el controlador principal de la aplicación y una serie de archivos con servicios y configuraciones generales.

Gráfico 38: estructura de ficheros del panel de administración 3

El archivo app.controller.js contiene el controlador principal de la aplicación AppCtrl, que contiene opciones de configuración de slim. Las únicas modificación que se ha realizado sobre este archivo han sido

```
Categories.query().$promise.then(
        function(payload) {
          $scope.nav.categories=payload;
      },
        function(errorPayload) {
            logger.logError('Se ha producido un error al obtener las categorías');
      });
```

El servicio Categories contiene las funciones de comunicación con el servidor para la gestión de las categorías. En el código de arriba usamos el método query para obtener la lista de categorías y poder mostrarla en el menú lateral.

```
$scope.logout=function(){
  userAuthFactory.logout()
}
```

La función logout se invoca desde el navbar e invoca al servicio método logout del servicio userAuthFactory. Este método elimina los datos del usuario dl almacenamiento local y redirige al estado de login.

En app.directives.js simplemente se define la directiva goBack que permite volver al estado anterior y en config.route.js se definen la ruta por defecto y la página de error 404.

```
$urlRouterProvider
      .otherwise('/dashboard');
  $stateProvider
      .state('404',{url:'/404',templateUrl: 'app/page/404.html'});
```

En data.factory.js definimos un servicio para gestionar la configuración del servidor REST.

- **dataFactory.restUrl** devuelve la url del servidor REST

Al usar un servicio nos permite definir la configuración en un único punto de manera que en caso de cambio de servidor sólo tendríamos que modificar este archivo. También nos permite usar url diferentes en función de si estamos ejecutando un servidor local o la versión de producción.

```
var restServer=function(){
        if($location.host()=="localhost"){
            return "http://localhost:3001/"
        }else{
            return "http://----t:3001/"
        }
    };
```

El archivo i118n.js contiene la configuración por defecto para la gestión de idiomas y el controlador LangCtrl para el selector de idiomas de la cabecera.

La carpeta **layout** contiene los archivos necesarios para la cabecera y la barra lateral. La mayoría de estos archivos no han sido modificados respecto a los originales de slim. Tan sólo se ha definido una nueva directiva en nav.directive.js.

```
angular.module('app.nav')
    .directive('toggleNavCollapsedMin', ['$rootScope', toggleNavCollapsedMin])
    .directive('collapseNav', collapseNav)
    .directive('highlightActive', highlightActive)
    .directive('toggleOffCanvas', toggleOffCanvas)
    .directive('navCategories', navCategories);
```

La directiva navCategories obtiene la lista de categorías del servidor las muestra con un enlace a su respectivo estado.

```
var directive={
  restrict: 'A',
  templateUrl: 'app/layout/navCategories.directive.html',
  controller: controller,
  controllerAs: 'nc',
```

```
  link: link
}
        ...
Categories.query().$promise.then(
          function(payload) {
            nc.categories=payload;
          },
          function(errorPayload) {
              logger.logError('Se ha producido un error al obtener las categorías');
          });
```

La plantilla correspondiente a esta directiva es:

```
<a href="#/ui"><i class="ti-wand"></i><span data-translate="CONTENT"></span></a>
<ul>
  <li ng-repeat="category in nc.categories"><a href="#/ui/typography">
    <i class="ti-angle-right"></i><span data-translate="TYPOGRAPHY"></span></a></li>

</ul>
```

En esta porción de código observamos algunos conceptos nuevos de nuestra aplicación. Por un lado tenemos la directiva data-translate que la empleamos para traducir el contenido del panel de control. Más adelante entraremos en más detalle sobre el funcionamiento de esta directiva.

La directiva ng-repeat de angularJs nos permite repetir un elemento html en función de los elementos de una colección. En este caso mostraremos un link por cada categoría en nc.categories.

Gráfico 39: estructura de ficheros del panel de administración 4

El estado dashboard muestra una serie de estadísticas del contenido y usuarios del sistema. Para los gráficos se ha usado el módulo angular-echarts. En el controlador se obtienen los datos de usuarios categorías y contenido y el template se muestran por medio de la directiva data-echarts.

```
<div data-echarts data-options="vm.pie1.options" style="height: 350px;"></div>
```

Los estados para la gestión de categorías, contenido y usuarios tienen una estructura similar. Vamos a utilizar la gestión de usuarios como ejemplo para explicar detalladamente el funcionamiento y posteriormente se explicará las particularidades de las otras dos secciones: categorías y contenido.

Gráfico 40: estructura de ficheros del panel de administración 5

En el archivo user.controller.js se define un bloque de configuración con las rutas para las operaciones crud (listado, edición, creación) y un controlador para cada una de las rutas.

```
angular.module('app.user', [])
    .config(config)
    .controller('userListCtrl', userListCtrl)
    .controller('userUpdateCtrl', userUpdateCtrl )
    .controller('userCreateCtrl', userCreateCtrl );

function config($stateProvider){
    $stateProvider
    .state('user',{
        abstract: true,
        url:'/users',
        template: '<ui-view/>'
    })
    .state('user.list',{
      url:'/list/:category_id',
        templateUrl:'app/user/user.list.html',
```

```
        controller:'userListCtrl',
        controllerAs:'vm'
    })

    .state('user.edit',{
        url:'/edit/:id',
        templateUrl:'app/user/user.form.html',
        controller:'userUpdateCtrl',
        controllerAs: 'vm'
    })

    .state('user.new',{
        url:'/new',
        templateUrl:'app/user/user.form.html',
        controller:'userCreateCtrl',
        controllerAs: 'vm'
    });
}
```

El controlador para el estado user.list **userListCtrl** obtiene la lista de usuarios del servidor y configura todas las opciones de ordenación , paginación y para el filtro de búsqueda del listado.

```
function loadUsers(){
    Users.query()
        .$promise
         .then( getSuccess,getError);
  }



function getSuccess(payload){
    var init=init;
    vm.contentItems=payload;

                ...

    vm.search = function() {
      vm.filteredItems = $filter('filter')(vm.contentItems, vm.searchKeywords);
      return vm.onFilterChange();
    };

       ...
```

```
function deleteUser(user){
   var id=user._id;

   var modalInstance = $uibModal.open({
      animation: false,
      templateUrl: 'app/modal/modal.confirm.html',
      controller: modalInstanceController,
      size: 'sm'
});

function modalInstanceController($scope,$uibModalInstance){
    $scope.title="Eliminar Usuario";
    $scope.message="¿Está seguro de que desea eliminar el usuario?";

    $scope.ok=function(){
       User.delete({id: id}).$promise.then(
       function(payload) {
         $uibModalInstance.close();
         $state.go($state.current.name, {}, {reload: true});
         logger.logSuccess('El usuario ha sido eliminado');
       },
       function(errorPayload) {
          logger.logError('Se ha producido un error al eliminar el usuario ');
       }
      );
    }

    $scope.cancel=function(){
      $uibModalInstance.dismiss("cancel");
    }
  }
};
```

La función para eliminar un usuario antes de proceder a eliminar el usuario muestra una ventana modal de confirmación.

La plantilla para el listado es una tabla html con una directiva ng-repeat por cada una de las categorías que cumplan con el requisito del filtro de búsqueda.

```
<tr data-ng-repeat="item in vm.filteredItems">
  <td>      {{item.name}}    </td>
```

```
    <td>      {{item.email}}   </td>
    <td>      {{item.admin}}   </td>
    <td width="100">
      <a
        href="javascript:;"
        class="btn-icon btn-icon-sm bg-info"
        ui-sref="user.edit({id:item._id})">
        <span class="glyphicon glyphicon-pencil"></span>
      </a>
      <a
        href="javascript:;"
        ng-if="vm.loggedUser!=item.email"
        class="btn-icon btn-icon-sm bg-danger"
        ng-click="vm.delete(item)">
        <span class="glyphicon glyphicon-minus-sign"></span>
      </a>
    </td>
  </tr>
```

Cada fila de la tabla tiene unos botones para editar y eliminar el contenido. El filtro de búsqueda se define en un input de tipo text que con cada presión de tecla llama a la fucnión vm.search definida en el controlador.

```
<input type="text"
    placeholder="{{'SEARCH' | translate}}..."
    class="form-control"
    data-ng-model="vm.searchKeywords"
    data-ng-keyup="vm.search()">
```

Los estados para la creación y la edición de usuarios comparten la misma plantilla. La plantilla hace uso de la directiva ng-if de angularJs para personalizar el aspecto y comportamiento del formulario en función de si se trata de edición con creación de usuarios.

```
<input ng-if="vm.showPwd || vm.isNew" ng-model="vm.newPwd">
<span class="show-password"
    ng-click="vm.showPwd=true"
    ng-if="!vm.showPwd"
    data-translate="PWD_CHANGE">
</span>
```

En el código anterior se usan las variables **vm.showPwd y mv.isNew** para controlar si se muestra el input para introducir la contraseña. En la creación de usuarios el input se debe

mostrar (vm.isnew=true) mientras que en la edición de usuarios inicialmente el input no se muestra sino que en su lugar se muestra un elemento span que al hacer click cambia el valor de **vm.showPwd** para que se muestre la contraseña.

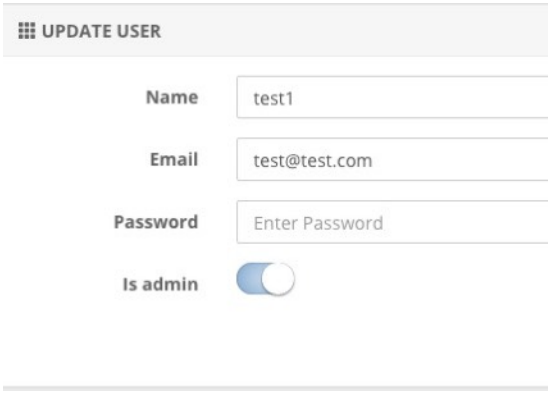


Gráfico 41: Formulario update user

En la plantilla se define un enlace de datos con el objeto vm.user mediante la directiva ng-model en cada uno de los inputs del formulario.

El controlador del estado **user.new** hace uso del método create del servicio User para enviar una petición de creación de usuario al servidor REST.

El controlador del estado **user.edit** obtiene los datos del usuario seleccionado haciendo uso del servicio **User.get** y los almacena en el objeto vm.user. Al hacer clic en el botón de actualizar envía la petición de actualización al servidor REST **User.update**.

```
User.get({id: $stateParams.id})
 .$promise
  .then(getUserSuccess,getUserError);

...

User.update({id:vm.user._id},data)
.$promise
  .then(...)
```

En el archivo **user.service.js** se define el servicio User para la conexión con el servidor REST. En este servicio se definen métodos para realizar las tareas CRUD sobre usuarios.

```
(function() {
	'use strict';

	angular.module('app.user')
		.factory('Users', getAllUsers)
		.factory('User', crud)


	//Obtenemos la lista de usarios
	function getAllUsers($resource, dataFactory){
		return $resource(dataFactory.restUrl()+'users/', {}, {
			query: { method: 'GET', isArray: true }
		});
	}
	//Crud
	function crud($resource, dataFactory){
		return $resource(dataFactory.restUrl()+'users/:id', {}, {
			create: { method: 'POST' },
			get: { method: 'GET',  },
			update: { method: 'PUT', params: {id: '@id'} },
			delete: { method: 'DELETE', params: {id: '@id'} }
		});
	}

})();
```

Para implementar las operaciones de administración de usuarios, categorías y contenido se ha empleado el servicio $resource de angularJS. $resource está desarrollado sobre el servicio $http y permite interactuar con apis RESTfull de manera sencilla y rápida para implementar las operaciones CRUD.

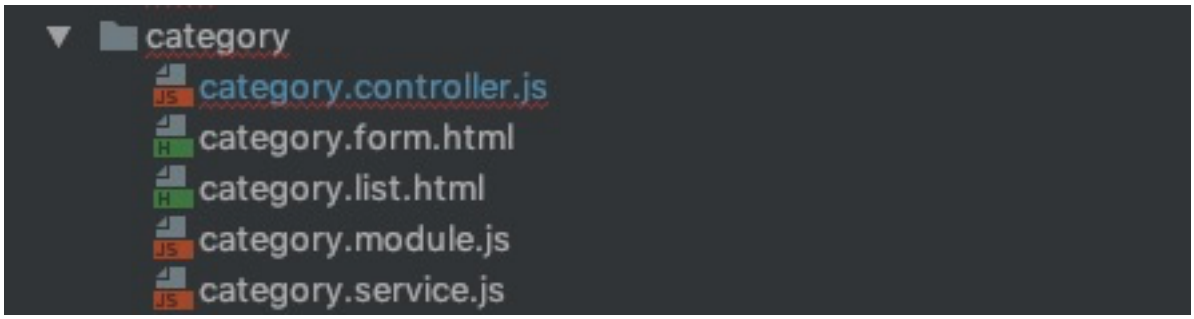


Gráfico 42: estructura de ficheros del panel de administración 6

Como se puede observar la estructura del directorio **category** es muy parecida a la de users, por tanto nos vamos a centrar en explicar simplemente las diferencias.

En el archivo **category.service.js** a parte de los métodos para las operaciones crud sobre las categorías se ha añadido un método para obtener los tipos de campos (fieldTypes) definidos en el servidor.

```
    //Obtenemos la lista de tipos de fieldTypes
function getAllFieldTypes($resource, dataFactory){
        return $resource(dataFactory.restUrl()+'fieldTypes/', {}, {
                query: { method: 'GET', isArray: true }
        });
}
```

En la plantilla para la creación y edición de categorías se muestra una sección en la que el usuario puede añadir los componentes que conformarán dicha categoría.

```
<div class="col-sm-12 callout callout-info"
  ng-repeat="component in vm.category.components track by $index"
  ng-init="componentIndex=$index">
...
</div>
```

En el controlador definimos una nueva directiva initFieldTypes que modifica el contenido del contenido del formulario del componente en función del tipo de fieldType seleccionado.

Por ejemplo si elegimos como tipo “campo de texto” se mostrarán las opciones por defecto, mientras que si cambiamos a “Enlace a URL o Fileupload” se mostrará un campo adicional para los tipos mime aceptados

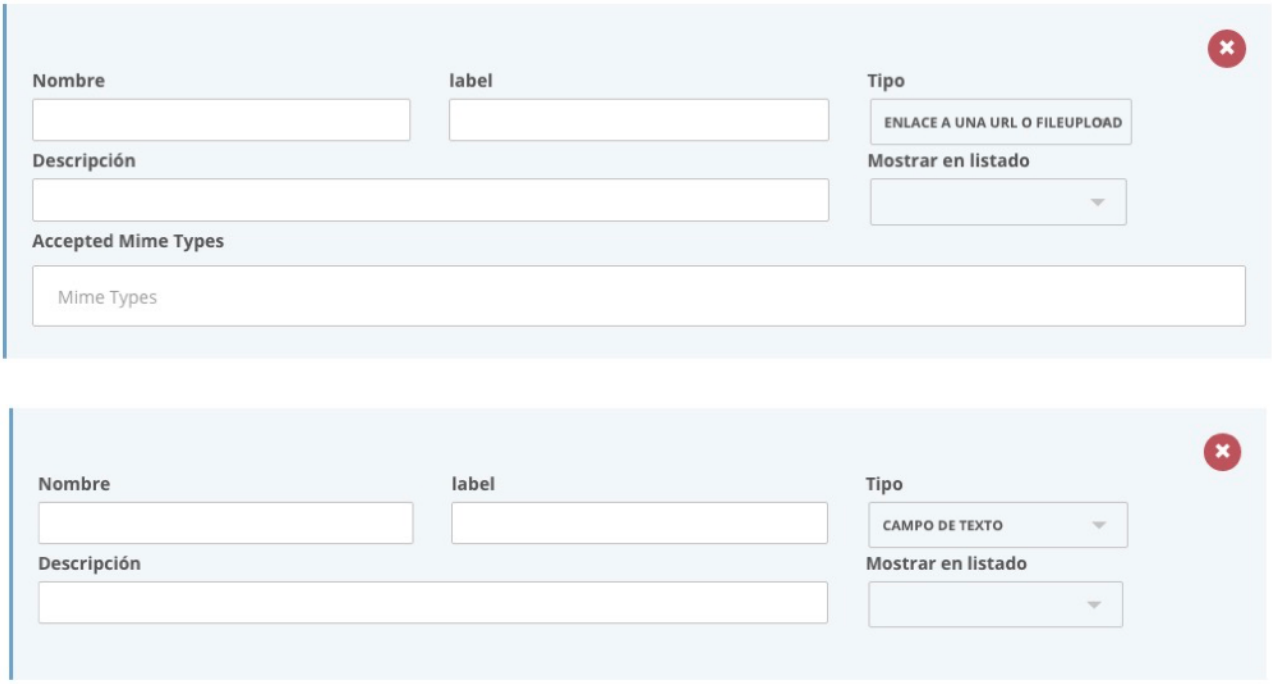


Gráfico 43:Formulario field types

La estructura básica del bloque de componentes de la plantilla para el formualrio de creación y edición de categorías es

```
<div class="col-sm-12 callout callout-info"
    ng-repeat="component in vm.category.components track by $index"
    ng-init="componentIndex=$index">
   ...
   <div class="col-sm-3">
      <label >Tipo</label>
      <span class="ui-select">
         <select
            ng-model="vm.category.components[componentIndex].type_id"
            ng-change="vm.setFieldType(componentIndex)">
               <option
                  component-index={{componentIndex}}
                  init-field-types
                  ng-repeat="item in vm.fieldTypes track by $index"
                  value="{{item._id}}"
                  ng-init="fieldTypeIndex=$index"
                  ng-selected="item._id==vm.category.components[componentIndex].type_id">
                  {{item.desc}}
               </option>
         </select>
      </span>
   </div>
```

```
...
<div class="row" ng-repeat="item in vm.componentOptions[componentIndex]">
  <div class="col-sm-12" >
    <label data-traslatem.label}}">{{item.label}}</label>
    <tags-input
      ng-model="vm.category.components[componentIndex]['extra'][item.name]"
      class="ui-tags-input"
      placeholder="{{item.desc}}"
      ng-if="item.type=='TAGS INPUT'">
      <auto-complete source="vm.loadMimeTypes($query,item)"></auto-complete>
    </tags-input>
  </div>
</div>
</div>
```

Como se puede observar el bloque se repite mediante un **ng-repeat** por cada componente definido en la categoría. En el select de tipo de componente se incluye la directiva **init-field-types** que actualiza las opciones extras mostradas en el bloque final.

Gráfico 44: estructura de ficheros del panel de administración 7

Nuevamente la estructura de directorios para la gestión de contenido es muy similar a las de categorías y usuarios, por tanto vamos a ir directamente a las particularidades de esta sección.

Como se puede observar hay dos archivos nuevos content.directives.js y content.linkFile.html.

En la plantilla **content.form.html** para los estados **content.new** y **content.edit** en la zona superior se muestra un selector de categorías en el que se incluye la directiva cms-category-select.

```
<select
  class="form-control"
  cms-category-select
  ng-model="vm.category_id"
  ng-options="item._id as item.name  for item in vm.categories" />

...

<div id="components-wrapper"></div>
```

La directiva **cms-category-select** actualiza el contenido del elemento html con id components-wrapper para que refleje los componentes de la categoría seleccionada.

En **conten.directives.js** se definen directivas para cada uno de los tipos de contenido disponibles en el sistema.

```
var directives=angular.module('app.content')
   .directive('fileModel', fileModel)
   .directive('cmsTextInput', cmsTextInput)
   .directive('cmsLinkfile',cmsLinkfile)
   .directive('cmsTextarea',cmsTextarea)
   .directive('cmsCategorySelect',cmsCategorySelect);
```

FileModel permite subir archivos al servidor.

```
function fileModel ($parse) {
   return {
      restrict: 'A',
      link: function(scope, element, attrs) {
         var model = $parse(attrs.fileModel);
         var modelSetter = model.assign;

         element.bind('change', function(){
            scope.$apply(function(){
               modelSetter(scope, element[0].files[0]);
            });
         });
      }
   };
 }
```

CmsTextInput es la directiva para los inputs de tipo texto

```
function cmsTextInput($compile,$parse){
    var directive={
      restrict: 'E',
      replace: true,
      transclude: false
    };

    directive.link = function(scope, element, attrs){
       var model = attrs.ngModel;
       var html = '<input type="text"  class="form-control" ng-model="'+model+'"/>';
       var newElem = $compile(html)(scope);
       element.replaceWith(newElem);
    }
    return directive;
 }
```

CmsLinkFile muestra un file uplad o un input de tipo texto

```
function cmsLinkfile($compile,$parse,$timeout,$http){
    var directive={
      restrict: 'E',
      require: '?ngModel',
      transclude:true,
      scope:{
       'component':"="
      },
      link: linkFunction
    };
    return directive;

    function linkFunction(scope,element,attrs){
      console.log(scope.component);
      if(scope.component.type_id){
        scope.mimetypes=getMimeTypes(scope.component.type_id.opts[0].options);
      }


      $http.get('app/content/content.linkFile.html').then(getSuccess);

      function getSuccess(response){
        var newElement=$(response.data);
        newElement.find('input').attr('ng-model',attrs.ngModel);
```

```
            newElement = $compile(newElement[0].innerHTML)(scope.$parent);
            element.replaceWith(newElement);
        }

        function getMimeTypes(data){
         var mimeTypes="";
         for(var i=0; i<data.length; i++){
           if(i>0){
             mimeTypes+=',';
           }
           mimeTypes+=data[i].text;
         }
         return mimeTypes;
        }

    }
}
```

CmsTextArea muestra un editor de texto enriquecido

```
        function cmsTextarea($compile,$parse){
            var directive={
              restrict: 'E',
              replace: true,
              transclude: false
            };

            directive.link = function($scope, element, attrs){
               var model = attrs.ngModel;
               var html = [
                  '<div',
                    'text-angular',
                    'class="ui-editor"',
                    'ng-model="'+model+'">',
                  '</div>',
                ].join(' ');

               $scope.editorOptions={
                  encoded: false,
                   tools: [
                              {
                        name: "maximize",
                        tooltip: "Maximize",
                        exec: function (e) {
```

```
                    var editor = $(this).data("kendoEditor");
                    var parent = editor.wrapper.parent();

                    parent.addClass('fullScreen');
                    editor.wrapper.css("height", parent.height());
                }
            },
            {
                name: "restore",
                tooltip: "Restore",
                exec: function (e) {
                    var editor = $(this).data("kendoEditor");
                    var parent = editor.wrapper.parent();
                    parent.removeClass('fullScreen');
                    editor.wrapper.css("height", 200);
                }
            },
            "bold", "italic", "underline","createLink","unlink",
            "fontSize",
            "justifyLeft", "justifyCenter", "justifyRight", "justifyFull",
            "insertUnorderedList","insertOrderedList", "indent", "outdent",
            "createTable", "addColumnLeft", "addColumnRight", "addRowAbove",
             "addRowBelow", "deleteRow", "deleteColumn",
            "cleanFormatting",
            "formatting",
            "insertHtml",
            "viewHtml",
          ],
        stylesheets: [
          "/css/bootstrap.min.css",
          "/css/editor.css"
        ]
      }

      var newElem = $compile(html)($scope);
      element.replaceWith(newElem);

    }
    return directive;
}
```

Por último está la directiva cmsCategorySelect que muestra un select con todas las categorías disponibles.

```
function cmsCategorySelect($compile,$parse,$filter, Categories){
    var directive={
      restrict: 'A',
      transclude: false
    };
    directive.link = function($scope, element, attributes){
          $scope.$watch('vm.category_id',function(newVal){
           if(newVal) {
             var category= $filter('filter')($scope.vm.categories, {_id:newVal},true);
             var wrap=element.closest("form").find("#components-wrapper");
             wrap.empty();
             for (var j in category[0].components){
                var component= category[0].components[j];
                var fieldName=component.type_id.name;
                var html='<div class="form-group"  ><label class="col-sm-2 control-
label">'+component.label+'</label><div class="col-sm-10" > <'+fieldName+'
component="vm.component" ng-model="vm.components.'+component.name+'"></div></div>';
                $scope.vm.component=component;
                var newElement=$compile(html)($scope);
                wrap.append(newElement);
             }
           }
         });

        return directive;
    }
```

En angular.js las directivas que manipulan el DOM normalmente usan la opción link para registrar los "listener" así como para actualizar el DOM. Se ejecuta después de que la plantilla haya sido clonada y es donde se debe incluir la lógica de la directiva.

En nuestra función link registramos un watcher para la variable vm.category_id. De manera que cada vez que se actualice la categoría se reconstruye el bloque del formulario correspondiente a los componentes de la categoría.

El archivo **content.linkFile.js** es la plantilla para la directiva linkFile.

5. Temporalización

Para el desarrollo de este proyecto se ha optado por el uso de una metodología de desarrollo ágil. Cuando se habla de desarrollo de software ágil siempre se habla de Scrum y en menor medida de Kanban.

Scrum y Kanban son herramientas de proceso que te ayudan a trabajar más eficazmente.

Herramienta = cualquier cosa usada como medio para realizar una tarea o propósito

Proceso = cómo trabajas.

Scrum en pocas palabras

- Organiza el equipo en pequeños equipos autogestionados y multidisciplinarios.
- Organiza el trabajo en una lista de entregables pequeños y concretos.
- Organiza el tiempo en iteraciones cortas, con código demostrable, al terminar cada iteración
- Optimiza la planificación de versiones y actualiza las prioridades en colaboración con el cliente
- Optimiza el proceso en cada iteración

Kanban en pocas palabras

- Visualiza el flujo de trabajo: al separar el trabajo en asuntos o issues, escribir cada asunto en una tarjeta de la pizarra Kanban y utilizar columnas que ilustran el estado de la tarjeta.

- Limita el trabajo en progreso: es recomendable asignar explícitamente cuantos ítems podemos tener en cada estado, un estado es una columna de la pizarra de Kanban.

- Mide el tiempo de iniciación de un asunto: con el objetivo de optimizar ese tiempo

Ambos procesos están basados en el desarrollo incremental pero Scrum es más restrictivo que Kanban ya que prescribe cosas como iteraciones y equipos interdisciplinarios. Por tanto Kanban es la solución que mejor se ajusta a las características de este proyecto.

Para planificar y supervisar el desarrollo del proyecto se ha optado por usar la herramienta software JIRA.

JIRA es una aplicación basada web para el seguimiento de errores, de incidencias y para la gestión operativa de proyectos aunque también se puede utilizar en áreas no técnicas para la administración de tareas. La herramienta fue desarrollada y está mantenida por la empresa australiana Atlassian.

Todos los fundamentos de Jira se agrupan alrededor del concepto básico de asunto o issue. Veamos a continuación estos conceptos de forma introductoria.

Un **asunto** es como una unidad de trabajo en la que alguien puede describir acciones realizadas o imputar trabajo. Si el sistema se utiliza como un gestor de incidencias, un asunto sería sinónimo de un ticket, pero Jira es mucho más. Un asunto puede usarse tanto para identificar errores, como requisitos, mejoras, reuniones, conocimientos, etc. Esto es lo que se conoce como tipo de asunto.

Un asunto puede verse como una bitácora de acciones a realizar que comparten varios usuarios (posiblemente clientes y desarrolladores) para resolver una determinada tarea. Alguien lo creó y alguien lo tiene asignado.

A parte de la información asociada al asunto, que variará en función del tipo de asunto, todo asunto tiene asociado un estado concreto (abierto, en progreso, cerrado, etc.). Los cambios de estado vienen definidos como flujos de trabajo.

El funcionamiento básico de Jira consiste en crear, buscar, y cambiar el estado de los asuntos en los que uno está trabajando.

Los asuntos se agrupan en lo que se conoce como **proyectos**. A más alto nivel, los proyectos se pueden agrupar en categorías y a más bajo nivel un proyecto puede dividirse en lo que se conocen como **componentes** o **versiones**. Los componentes actúan como etiquetas de los asuntos. Las versiones representan hitos de entrega.

Gráfico 45: Componentes del proyecto en Jira

Los tipos de asuntos que vamos a manejar en este proyecto son:

- **Story**: indica un requerimiento de software expresado de manera breve y generalmente usando un vocabulario no técnico.
- **Task**: se trata de una unidad de trabajo básica, generalmente como parte de una story
- **Epic**: engloba trabajos de cierta envergadura, se puede entender como una Story compleja que puede ser dividida en varias stories.
- **Bug**: Representa un problema en el funcionamiento del producto.
- **SubTask**: Las tareas complejas pueden dividirse en subtareas más sencillas

A continuación se detallan las asuntos que se crearon para cada uno de los componentes así como el tiempo empleado en cada uno de ellos

Componente	Asuntos	Descripción	Tiempo
Análisis de Requisitos	3 Issues	Captura, Análisis y Especificación de requisitos	130 h
Documentación	1 Issue	Redacción de la documentación del proyecto	166 h
Interfaz Gráfica	11 Issues	Desarrollo de interfaz Gráfica para la administración de contenidos	229 h
Prueba	2 Issues	Prueba y mantenimiento del software	165 h
Rest Api	13 Issues	Desarrollo de API REST para la gestión de contenido	115 h
		Total	**805 h**

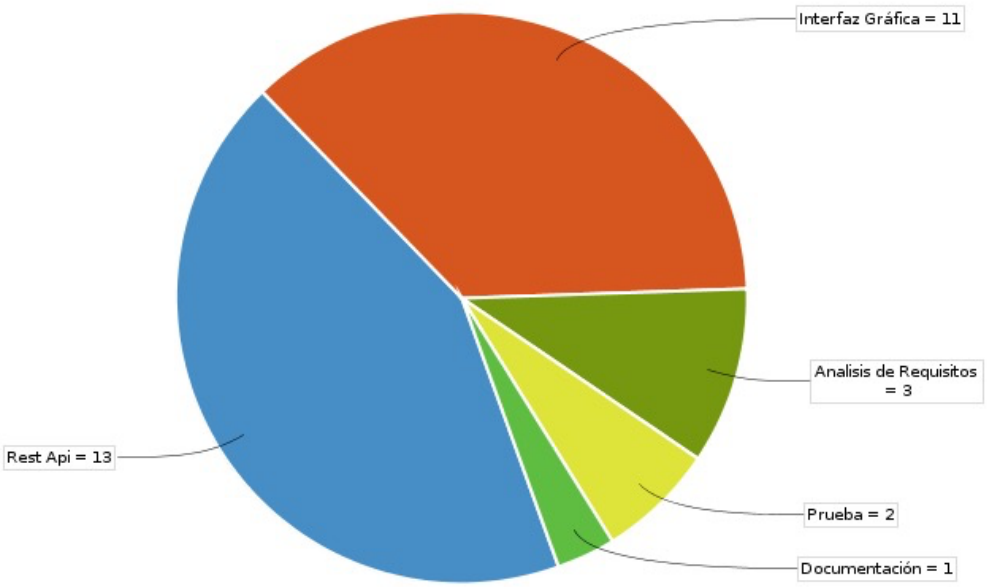


Gráfico 46: Componentes y asuntos del proyecto

Análisis de Requisitos

T	Key	Components	Summary	P	Status	Resolution	Time Spent	
	PJR-15	Analisis de Requisitos	Especificación de requisitos	↑	DONE	Done	40h	...
	PJR-14	Analisis de Requisitos	Análisis de requisitos	↑	DONE	Done	60h	
	PJR-13	Analisis de Requisitos	Captura de requisitos	↑	DONE	Done	30h	

Api Rest

T	Key	Components	Summary	P	Status	Resolution	Time Spent	
	PJR-20	Rest Api	Implementación de la capa de acceso a Datos	↑	DONE	Done	35h	
	PJR-19	Rest Api	Implemetación de la Capa de negocios	↑	DONE	Done	40h	
	PJR-18	Rest Api	Autéticación Api Rest	↑	DONE	Done	30h	
	PJR-17	Rest Api	Desarrollo de la aplicación node para la api REST	↑	DONE	Done		
	PJR-16	Rest Api	Estrutura Básica de la aplicación	↑	DONE	Done	33h	...
	PJR-12	Interfaz Gráfica, Rest Api	Instalar Node local	↑	DONE	Done	4h	
	PJR-11	Interfaz Gráfica, Rest Api	Instalar MongoBd local	↑	DONE	Done	5h	
	PJR-6	Rest Api	Instalación de PM2	↑	DONE	Done	1h	
	PJR-5	Rest Api	Instalación Node	↑	DONE	Done	2.5h	
	PJR-4	Rest Api	Instalación de Git y configuración de Git	↑	DONE	Done	1h	
	PJR-3	Rest Api	Instalación mongo	↑	DONE	Done	0.5h	
	PJR-2	Rest Api	Configuración servidor Api REst	↑	DONE	Done		
	PJR-1	Rest Api	Definición del modelo de datos	↑	DONE	Done	20h	

Interfaz gráfica

T	Key	Components	Summary	P	Status	Resolution	Time Spent	
	PJR-26	Interfaz Gráfica	Modulo de contenido de la interfaz gráfica	↑	DONE	Done	80h	...
	PJR-25	Interfaz Gráfica	Modulo categorías de la interfaz grafica	↑	DONE	Done	30h	
	PJR-24	Interfaz Gráfica	Modulo de usuario de la interfaz gráfica	↑	DONE	Done	26h	
	PJR-23	Interfaz Gráfica	Modulo principal de la interfaz gráfica	↑	DONE	Done	29h	
	PJR-22	Interfaz Gráfica	Autenticación del la interfaz gráfica	↑	DONE	Done	24h	
	PJR-12	Interfaz Gráfica, Rest Api	Instalar Node local	↑	DONE	Done	4h	
	PJR-11	Interfaz Gráfica, Rest Api	Instalar MongoBd local	↑	DONE	Done	5h	
	PJR-10	Interfaz Gráfica	Configurar ngnix	↑	DONE	Done	8h	...
	PJR-9	Interfaz Gráfica	Instalar Slim	↑	DONE	Done	3.5h	
	PJR-8	Interfaz Gráfica	Instalar Ngnix	↑	DONE	Done	1.5h	
	PJR-7	Interfaz Gráfica	Configuración de entorno desarrollo	↑	DONE	Done	18h	

1–11 of 11

Pruebas

T	Key	Components	Summary	P	Status	Resolution	Time Spent	
	PJR-28	Prueba	Pruebas de la interfaz Gráfica	↑	DONE	Done	90h	...
	PJR-27	Prueba	Pruebas de la Api Rest	↑	DONE	Done	60h	

Documentación

	PJR-21	Documentación	Documentación	↑	IN PROGRESS	*Unresolved*	115h	...

6. Prueba y Mantenimiento.

Durante el desarrollo del proyecto se han llevado a cabo una serie de pruebas para comprobar el correcto funcionamiento del sistema y la seguridad e integridad del mismo.

Durante la fase de implementación se ha diseñado una serie de test, permitiendo producir resultados satisfactorios y garantizando el correcto funcionamiento de todos los recursos que se han ido desarrollando.

6.1 Pruebas Unitarias y Funcionales.

Las pruebas unitarias aseguran que un único componente de la aplicación produce una salida correcta para una determinada entrada. Este tipo de pruebas validan la forma en la que las funciones y métodos trabajan en cada caso particular.

Las pruebas funcionales no solo validan la transformación de una entrada en una salida, sino que validan una característica completa.

Una vez definidos los recursos de nuestra API se ha empleado la herramienta postman para comprobar el correcto funcionamiento de cada uno ellos, comprobando que acepta las peticiones diseñadas y devuelve las respuestas correctas.

En cada una de estas pruebas se han asegurado el correcto funcionamiento del sistema de seguridad, para ello se han simulado peticiones HTTP con las cabeceras de autorización tanto para un usuario válido del sistema como para un usuario sin privilegios de acceso. Para cada uno de los recursos de nuestra API se han simulado peticiones HTTP con datos válidos y con datos alterados para comprobar la respuesta del sistema ante peticiones inesperadas, evitar duplicidad de datos, formato de información incorrecto, etc.

Para la interfaz gráfica de administración las pruebas unitarias se llevaron a cabo mediante la comprobación del correcto funcionamiento en el manejo de cada recurso de manera individual, la presentación de los datos y el correcto flujo de actividad entre pantallas.

6.2 Pruebas de Integración.

Las pruebas de integración consistieron en la comprobación del funcionamiento de todos los módulos como parte del sistema global. Por ello tras la creación de cada módulo o recurso se han realizado una serie de tests para confirmar su integración con el resto del sistema.

De especial importancia ha sido la comprobación de la integración de cada recurso con el módulo de seguridad para garantizar que los recursos privados sólo pueden ser accedidos haciendo uso de las correspondientes credenciales.

6.3 Pruebas de Sistema

Una vez finalizado el desarrollo de la API REST y de la interfaz de administración se han llevado a cabo una serie de pruebas de sistema para verificar el correcto funcionamiento del desarrollo en todo su conjunto.

Una vez finalizadas y aprobadas las pruebas del sistema en el entorno de desarrollo, el sistema está listo para su paso al entorno de producción, donde tendremos que repetir las mismas pruebas para garantizar su correcto funcionamiento

7. Análisis de costos y modelo de negocio.

7.1 Análisis de costos

Atendiendo al número de horas dedicadas a las tareas de análisis, diseño e implementación y pruebas, el coste de desarrollo del Sistema de Gestión de Contenido dinámico asciende a unos 9.000€. Este coste es totalmente fuera de mercado para pequeños clientes pero la idea detrás de este proyecto es la de crear una herramienta de fácil instalación, mantenimiento y ampliación que se pueda reutilizar para varios clientes.

7.2 modelo de explotación

El modelo de explotación de la herramienta se puede abordar desde dos modelos diferentes, como plataforma de Software como un Servicio (SaaS) o como software bajo licencia.

SaaS proviene del acrónimo inglés Software as a Service (Software como Servicio) y la característica que lo hace especial con respecto de otras opciones es que no requiere por parte del cliente ninguna instalación, ni de infraestructuras (servidores). La aplicación está "colgada" en internet y se ejecuta en el servidor del proveedor del servicio. En la práctica esto significa que pagando una cuota mensual (Servicio) dispones de la licencia de la aplicación y de todos los servicios.

Haciendo uso de este modelo y suponiendo una cuota media de 75€ por cliente y mes serían necesarias un total de 120 cuotas mensuales para amortizar los costes de programación. Si aparte de la interfaz de administración el cliente necesita que se le desarrolle una web pública que consuma los datos de la API REST se le debería cobrar una cuota de alta a determinar por la complejidad del diseño planteado.

A los costes de desarrollo hay que sumar los costes de infraestructura. Como ya se ha comentado anteriormente el sistema está pensado para escalar fácilmente por lo que se puede empezar con un servidor básico, en el mercado hay ofertas desde 20€/mes y escalar conforme crezca el número de clientes.

En el mercado actual hay ofertas que nos permiten dar servicio a más 200 clientes por 150€ al mes. Este cálculo se ha realizado teniendo en cuenta la contratación de 3 servidores, uno para la API REST Node, otro para la base de datos mongoDB y otro para la interfaz de cliente y web pública.

Se podría contratar un cuarto servidor para las copias de seguridad o bien usar los cada servidor como servidores de copias de seguridad de los otros.

La principal ventaja para los clientes de esta forma de explotación es que no tienen que preocuparse para nada de la parte técnica y que siempre tendrán a su disposición todas las actualizaciones.

La otra forma de explotación consiste en la venta de una licencia del software y que el cliente se encargue de administrar el servidor. Suponiendo un precio medio de 1500€ por licencia, con tan sólo 6 clientes ya se cubrirían los costes de programación.

8. Conclusiones y trabajos futuros

8.1 conclusiones

El proyecto tenía como objetivo crear un Sistema de Gestión de Contenido dinámico a través de un servicio web REST y una interfaz gráfica de administración.

El nuevo sistema desarrollado saca partido de las ventajas que ofrece la pila de aplicaciones MEAN para solventar los problemas que presentaban los Sistemas de gestión de contenidos basados en la pila de aplicaciones LAMP.

El desarrollo se ha llevado a cabo haciendo uso de una metodología ágil y se han llevado a cabo una serie de pruebas en un entorno de desarrollo local para garantizar su funcionamiento.

El sistema resultante está compuesto por:

- Una API REST basada en node y mongoDB que permita administrar todos los recursos necesarios para la gestión y presentación de contenido dinámico.

- Una herramienta de administración basada en web, fácil e intuitiva, de manera que cualquier usuario sin conocimientos avanzados de informática pueda gestionar el contenido de un sitio web.

Una vez superadas todas las pruebas el sistema se ha implantado con éxito en servidores de producción y está sirviendo contenido a sitios webs públicos.

8.2 Trabajos futuros.

Durante el plazo de ejecución de este proyecto ha sido lanzada la versión 2 de angularJS. Esta nueva versión supone un cambio sustancial con respecto a la versión anterior, hasta el punto de que parece un framework completamente nuevo, incluso su denominación ha pasado a ser Angular a secas en vez de angularJs. El coste de actualización de la interfaz

de cliente es demasiado elevado como para asumirlo como parte del proyecto por lo que queda pendiente como trabajo de mejora futura.

En el servidor node se hace uso de callbacks para gestionar las operaciones asíncronas. Tras estudiar diferentes opciones he considerado que es mejor cambiar a un patrón de programación asíncrona basado en promesas. Al ser una abstracción más avanzada, las promesas permiten operaciones como esperar a que diversas operaciones asíncronas terminen de ejecutarse de manera concurrente, mejoran la legibilidad del código y facilitan el manejo de errores. En definitiva, aportan ventajas competitivas frente a los callbacks.

En la actualidad el sistema de de gestión de contenido es capaz de gestionar tipos de contenido básicos, que cubren un porcentaje elevado de las necesidades de los clientes, en un futuro y con la llegada de nuevos clientes estos tipos de datos se podrán ampliar. El sistema ha sido desarrollado para facilitar esta clase de ampliaciones.

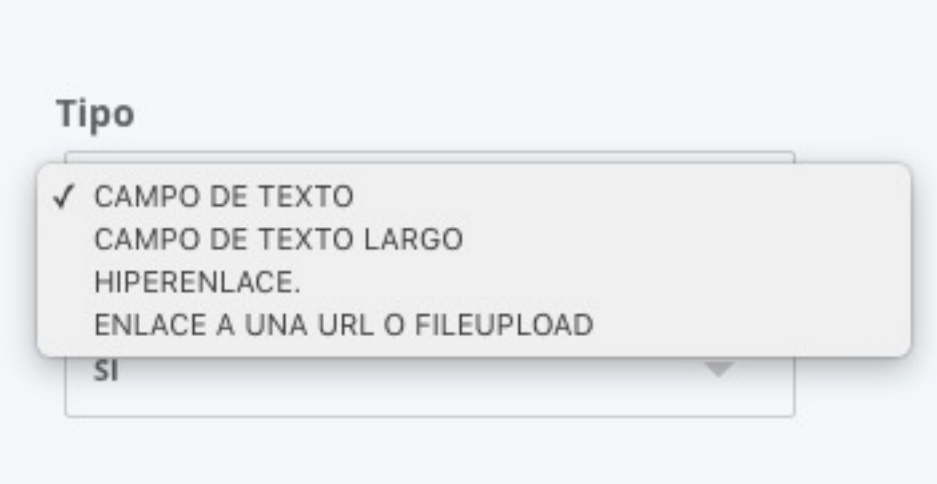


Gráfico 47: Tipos de contenido

Inicialmente el modelo de explotación pensado para la plataforma era la venta de licencias por lo que el sistema y el modelo de datos está pensado para gestionar contenido de un sólo cliente. Por lo que sea usado por distintos clientes se requerirá instalaciones independientes. A pesar de que la instalación es sencilla y rápida, resulta inviable para un modelo de negocio basado en Software como Un Servicio (SAAS y con un número de clientes elevados. Por tanto para poder explotarlo como SAAS habrá que realizar las

modificaciones pertinentes para convertirlo en multicliente e incluir en el propio sistema las funcionalidades para alta y gestión de clientes.

Referencias

Restfull Api Best Practices. (http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api)

The Basics of REST and RESTful API Development (http://www.hongkiat.com/blog/rest-restful-api-dev/)

Architecting a Secure RESTful Node.js app (http://thejackalofjavascript.com/architecting-a-restful-node-js-app/)

Install MongoDB Community Edition on OS X (https://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/)

Apache vs Nginx: Practical Considerations (https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations)

Understanding MVVM - A Guide For JavaScript Developers (https://addyosmani.com/blog/understanding-mvvm-a-guide-for-javascript-developers/)

NoSql data modeling technique (https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/)

AngularJs Style Guide: (https://github.com/johnpapa/angular-styleguide/tree/master/a1)